**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**
**Навчально-науковий фізико-технічний інститут**
**Кафедра математичного моделювання та аналізу даних**

«До захисту допущено»
Завідувач кафедри
_____ Іван ТЕРЕЩЕНКО
«__» _____ 2025 р.

# Дипломна робота
**на здобуття ступеня бакалавра**

зі спеціальності: 113 Прикладна математика
на тему: **«Моделювання оптимальної схеми розміщення авіаційного логістичного парку»**

Виконав:
студент 4 курсу, групи ФІ-11
Барабаш Дмитро Володимирович _____

Керівник:
канд. техн. наук, старший дослідник, доцент
Хайдуров Владислав Володимирович _____

Рецензент:
доцент кафедри вищої та прикладної математики
НУБіП України, канд. фіз.-мат. наук, доцент
Цюпій Тамара Іванівна _____

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

Студент _____

Київ — 2025

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**
**Навчально-науковий фізико-технічний інститут**
**Кафедра математичного моделювання та аналізу даних**

Рівень вищої освіти — перший (бакалаврський)
Спеціальність — 113 Прикладна математика,
ОПП «Математичні методи моделювання, розпізнавання образів та комп'ютерного зору»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Іван ТЕРЕЩЕНКО
«__» _____ 2025 р.

## ЗАВДАННЯ
**на дипломну роботу**

Студент: Барабаш Дмитро Володимирович

1. Тема роботи: «*Моделювання оптимальної схеми розміщення авіаційного логістичного парку*», науковий керівник дисертації: канд. техн. наук, старший дослідник, доцент Хайдуров Владислав Володимирович, затверджені наказом по університету №1761-с від «26» травня 2025 р.

2. Термін подання студентом роботи: «__» _____ 2025 р.

3. Об'єкт дослідження: *Логістичні парки аеропортів, принцип функціонування логістичних парків, схема планування логістичного парку аеропорту*

4. Предмет дослідження: *Методи й алгоритми глобальної оптимізації для задач, що описуються складними математичними моделями з різними за складністю функціональними обмеженнями.*

5. Перелік завдань: *аналіз сучасного стану розвитку логістичного парку аеропорту, програмна реалізація математичної моделі схеми авіаційного логістичного парку та тестування методів оптимізації схеми розміщення.*

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: *Презентація доповіді*

7. Орієнтовний перелік публікацій: *XXIII Всеукраїнській науково-практичній конференції студентів, аспірантів та молодих вчених, секція «Математичне моделювання та аналіз даних»*

8. Дата видачі завдання: 10 вересня 2024 р.

Календарний план

| № з/п | Назва етапів виконання дипломної роботи | Термін виконання | Примітка |
|---|---|---|---|
| 1 | Узгодження теми роботи із науковим керівником | Вересень 2024 р. | Виконано |
| 2 | Огляд опублікованих джерел стосовно сучасного стану логістичних парків аеропортів та відповідного математичного забезпечення побудови оптимальної схеми розміщення | Жовтень–грудень 2024 р. | Виконано |
| 3 | Огляд опублікованих джерел стосовно генетичного алгоритму, методу на основі рою частинок та їх модифікацій | Січень–лютий 2025 р. | Виконано |
| 4 | Програмна реалізація моделі побудови оптимальної схеми розміщення логістичного парку аеропорту | Березень–квітень 2025 р. | Виконано |
| 5 | Порівняльний аналіз роботи класичних й модифікованих алгоритмів | Квітень–травень 2025 р. | Виконано |
| 6 | Написання пояснювальної записки до дипломної роботи, створення презентації та захист дипломної роботи | Травень–червень 2025 р. | Виконано |

Студент _____ Дмитро БАРАБАШ

Керівник _____ Владислав ХАЙДУРОВ

# РЕФЕРАТ

Кваліфікаційна робота містить: 84 стор., 16 рисунків, 4 таблиці, 25 джерел.

Метою дослідження є побудова й програмна реалізація математичної моделі планування оптимальної схеми розміщення авіаційного логістичного парку, попередньо проаналізувавши сучасний стан розвитку логістичних парків, їх принцип функціонування та схему планування, з використанням методів й алгоритмів глобальної оптимізації.

Для моделювання було обрано два популяційні методи, а саме: генетичний алгоритм, та метод на основі рою частинок. Для них було запропоновано та розроблено ряд модифікацій, більшість з яких за результатами роботи мали кращий результат, ніж класичні методи за різними параметрами.

АВІАЦІЙНИЙ ЛОГІСТИЧНИЙ ПАРК, ГЕНЕТИЧНИЙ ОПТИМІЗАТОР, ОПТИМІЗАТОР НА ОСНОВІ РОЮ ЧАСТИНОК, ПЛАНУВАННЯ ОБ'ЄКТА

# ABSTRACT

The qualification work contains: 84 pages, 16 figures, 4 tables, 25 sources.

The purpose of the research is to build and programmatically implement a mathematical model for planning the optimal layout scheme of aviation logistics park, having previously analyzed the current state of development of logistics parks, their principle of operation and planning scheme, using global optimization methods and algorithms.

Two population methods were selected for modelling, namely: a genetic algorithm and a method based on a swarm of particles. A number of modifications were proposed and developed for them, most of which, according to the results of the work, had better result than classical methods for various parameters.

AVIATION LOGISTICS PARK, GENETIC OPTIMIZER, PARTICLE SWARM OPTIMIZER, FACILITY PLANNING

# ЗМІСТ

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

GA — генетичний алгоритм

PSO — оптимізатор на основі рою частинок

# ВСТУП

**Актуальність дослідження.** Сучасне планування логістичних парків аеропортів являє собою складну багатокритеріальну задачу, яка потребує врахування численних техніко-економічних факторів, такі як розміри пасажиропотоків, коливання цін на паливо, дотримання вимог до технічного та екологічного стану.

Популяційні алгоритми, такі як генетичний алгоритм чи метод на основі рою частинок, демонструють високу ефективність у вирішенні подібних оптимізаційних задач завдяки здатності ефективно досліджувати великомасштабні простори рішень в умовах обмеженої аналітичної інформації.

**Метою дослідження** є побудова й програмна реалізація математичної моделі планування оптимальної схеми розміщення авіаційного логістичного парку з використанням популяційних алгоритмів.

**Задача дослідження** полягає у наступному:

1) Провести огляд сучасного стану логістичних парків аеропортів, а також провести аналіз математичного забезпечення реалізації оптимальної схеми розміщення парку;

2) Проаналізувати методи й алгоритми вирішення проблеми пошуку глобальних екстремумів функцій / функціоналів з обмеженнями різної складності;

3) Програмно реалізувати математичну модель оптимальної схеми розміщення авіаційного логістичного парку кількома методами оптимізації.

*Об'єктом дослідження* є логістичні парки аеропортів, принцип функціонування логістичних парків, схема планування логістичного парку аеропорту.

*Предметом дослідження* є методи й алгоритми глобальної оптимізації для задач, що описуються складними математичними

моделями з різними за складністю функціональними обмеженнями.

При розв'язанні поставлених завдань в якості *методів дослідження* використовувався генетичний оптимізатор та оптимізатор на основі рою частинок.

**Наукова новизна** отриманих результатів полягає в описі та застосуванні нових модифікацій генетичного оптимізатора та оптимізатора на основі рою частинок для планування логістичних парків.

**Практичне значення** результатів полягає в програмній реалізації методів знаходження оптимальної схеми розміщення авіаційного логістичного парку. Отримані модтфікації мали кращі результати в порівнянні з класичними методами для реалізованої моделі.

**Апробація результатів та публікації**: XXIII Всеукраїнській науково-практичній конференції студентів, аспірантів та молодих вчених, секція «Математичне моделювання та аналіз даних».

# 1 ОГЛЯД СУЧАСНОГО СТАНУ ЛОГІСТИЧНИХ ПАРКІВ АЕРОПОРТІВ ТА ВІДПОВІДНЕ МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ ПОБУДОВИ ЙОГО ОПТИМАЛЬНОЇ СХЕМИ РОЗМІЩЕННЯ ОДНОГО ПАРКУ

Розділ присвячений опису з точки зору літератури тому, який загальний стан логістичних парків, чим логістичні парки аеропортів вирізняються з-поміж інших, та яку модель буде використано для планування цих об'єктів.

## 1.1 Сучасний стан і розвиток логістичних парків

Сучасна галузь логістики зазнає швидкого росту ще з кінця XX століття як потужний інструмент підтримки економічної глобалізації та промислової спеціалізації, під час якого вона відокремилася від логістичних відділів всередині різних компаній до повноцінного сектора послуг. Оскільки логістика слабко залежить від однієї галузі, то багато країн та регіонів прагнуть будувати та розвивати логістичні об'єкти, як логістичні кластери чи парки, щоб зменшити безробіття, замінити втрачені виробничі сектори та підтримати інші галузі [6].

Серед значущих об'єктів логістичної галузі є логістичні парки (рис. 1.1). Він не отримав загальноприйнятої термінології, і тому часто використовують інші назви: розподільчий парк, логістична основа [9] та інші. Логістичний парк можна визначити як логістичний збірний пункт в сільській місцевості або на перехресті між містом та селом поблизу основних транспортних коридорів, до складу якого входить транспортна система, інформаційна система та система управління логістикою. Основна мета логістичних парків полягає в тому, щоб діяти як проміжний пункт до віддалених районів, як правило, шляхом зберігання

товарів для первинних комерційних мереж розподілу та гіпермаркетів для подальшого розподілу в регіональному масштабі [9].

**(а)** **(б)**

**Рисунок 1.1** – Приклад логістичного парку: (а) в Німеччині, (б) в Японії

Логістичні парки класифікують за такими критеріями [21]:

– за потребами клієнтів: публічний, для самостійного використання;

– за типом сервісів: склад, розподільний центр, контейнерний майданчик, внутрішній контейнер, інтермодальний термінал, внутрішній порт, вантажне містечко, головний портовий термінал;

– за основними функціями: центр деконсолідації, центр відвантаження, розподільний центр, центр збору, резервний центр і центр обробки;

– за температурою: з нормальною температурою, з холодними ланцюгами.

Важливим фактором є степінь втручання держави в управління та інвестування і відповідно до цього розрізняють державні, приватні та під змішаним управління логістичні парки. Відповідно до досліджень, кожний вид може генерувати прибуток, однак найбільш ефективними є ті, які покладаються на більш приватне управління, але у свій перший етап планування та будівництва роль держави була ключовою [12].

На сьогодні дослідження про логістичні парки є багатими та глибокими, а теми дослідження в основному включають наступні категорії: дослідження розміщення логістичних парків, дослідження

режиму управління логістичного парку, дослідження з оцінки ефективності функціонування логістичного парку [21].

Крім безпосередньо економічної вигоди для регіону, логістичний парк створює розподіл ризиків, навантажень, знань, праці та інфраструктури, чим допомагає різним типам компаній ефективно співпрацювати та надавати кращі послуги за нижчими цінами; велика роль держави в розвитку логістичного кластера знижує бар'єри входу та виходу для логістичних фірм шляхом надання інфраструктури, фінансування та регулювання; створення логістичних парків є одним з потужних інструментів для залучення нових логістичних фірм [6]; логістичні парки за рахунок ефективного транспортування зменшують викиди вуглецю та забруднення повітря в регіоні [22].

Логістичні парки будують по всьому світу, але найбільш розвинулися та здобули досвіду будівництва Німеччина та Японія [21]. Ряд логістичних парків було побудовано в Україні, але розвиток цієї галузі залишався повільним. Також з початком війни логістична нерухомість зазнала значних збитків, деякі об'єкти були знищені, оскільки вони є стратегічно важливими об'єктами [5].

## 1.2 Характеристики та функції логістичного парку аеропорту

Логістичні парки аеропортів (рис. 1.2) в порівнянні з іншими парками, завжди будуються поблизу аеропортів, і тому вони в основному обслуговують високотехнологічні галузі, кур'єрські служби, компанії та галузі, орієнтовані на авіаційну промисловість, надаючи персоналізовані логістичні послуги та формуючи регіони кластерної логістики [23].

Також для даних логістичних парків властиві додаткові особливості. По-перше, розташування поруч з авіаційними вузлами та сполучення з іншими видами транспорту створює високоінтегровану

**Рисунок 1.2** – Логістичний парк аеропорту в Нідерландах

мережу, яка може надавати ще більш гнучкі та ефективні рішення. По-друге, внаслідок авіаційних перевезень парк обслуговує значною мірою дорогі товари з обмеженим терміном доставки або спеціальними умовами зберігання, як наприклад квіти, фрукти чи медикаменти. По-третє, сучасні авіаційні логістичні парки обладнані передовими інформаційними технологіями та системами автоматизації, включаючи програмне забезпечення для управління логістикою, автоматизовані склади та розумні системи сортування, які будуть оптимізувати швидкість і ефективність транспортування вантажів [23].

Для логістичних парків аеропортів виконуються всі притаманні для логістичних парків функції, а саме: транспортна функція, складська функція, вантажно-розвантажувальна функція, функція пакування, функція управління та обробки інформації. Окрім зазначених, сучасний логістичний парк також має такі додаткові функції: функцію розрахунків, прогнозування попиту, консультацій щодо навчання [21].

Також для логістичного парку аеропорту притаманні ще чотири функції. По-перше, це функція вантажного транзиту, тобто транзитні послуги, сортування та обробка вантажу для вантажних рейсів. По-друге, це додаткові інформаційні функції парку авіаційної логістики, як-от

надання інформації про вантажі та рейси чи правила перевезень для бізнесів [8]. По-третє, це функція перевірки безпеки, тобто усі товари проходять митне оформлення та перевірку на небезпечні матеріали, і у разі чого затримують всі вантажі, які не мають належної документації на перевезення небезпечних речовин. По-четверте, це надання послуг для екстрених служб чи допомога в надзвичайних ситуаціях: логістичні парки аеропортів можуть швидко мобілізувати ресурси з усієї країни, надаючи термінові логістичні послуги, щоб гарантувати оперативну доставку критично важливих товарів, мінімізуючи таким чином кількість жертв і економічні втрати [25].

## 1.3 Схема функціональних зон логістичного парку аеропорту

Функціональна зона є основною просторовою одиницею логістичного парку, кількість яких визначається в залежності від набору необхідних функцій. Функціональні зони повинні бути встановлені розумно, щоб усі робочі зони могли співпрацювати одна з одною для завершення логістичних операцій [14].

У літературі немає однозначних визначень до змісту функціональних зон логістичних парків, і вони частково довизначаються дослідниками, тому за основу і надалі в даній роботі буде розглядатись зони авіаційних логістичних парків, визначені в статтях [25, 20, 24]:

– Вантажна зона аеропорту: забезпечує прийом, доставку, обробку вантажів, тимчасове зберігання та інші зони аеропорту міста; включаючи громадські вантажні термінали аеропорту, базові вантажні станції авіакомпаній, платформи обробки вантажів, митний огляд і приймальні склади;

– Митна логістична зона: зона складається з митних складів та станцій перевірки та використовується для надання митних послуг для

імпорту та експорту товарів;

– Зона міжнародного транзиту та розподілу: зона призначена для відправки та наданні послуг з для сортування та розподілу товарів, які проходять транзитом;

– Зона обробки з доданою вартістю: включає допоміжні операції, такі як відокремлення та відсортування, оцінювання та класифікування, об'єднання та запакування, вимірювання та зважування, закодовування та зняття на плівку, перепакування тощо;

– Зона міжнародної торгівлі: ця зона надає послуги з зберігання та упакування для подальшого продажу міжнародних вантажів, які надходять чи залишають аеропорт;

– Комплексна допоміжна зона: зона включає обслуговування авіаційних, транспортних, логістичних компаній, водночас задоволення ділових і життєвих потреб персоналу та пасажирів за допомогою побудованого на території житла, магазинів, лікарень, стоянок тощо;

– Зона інформаційного обслуговування: надає інформації про логістику кожного складу, про вантажні рейси, оформлення митних декларацій, планування програми логістики, повідомлення та аналіз погодних умов та інші пов'язані інформаційні послуги.

Також деякі дослідники виділяють такі зони: науково-дослідницька зона [23], зона обробки спеціальних вантажів [15], зона холодного зберігання [8].

## 1.4 Математична модель оптимальної схеми розміщення авіаційного логістичного парку

У даній роботі математична модель схеми планування буде базуватись на моделі, що описана у статті [4]. Відповідно, будемо розглядати прямокутний авіаційний логістичний парк, який сіткою буде розділено на однакові прямокутні клітинки, і будемо припускати, що одна

клітинка належить тільки одній функціональній області (рис. 1.3). Також розраховуємо кількість необхідних клітинок для кожної функціональної зони так, щоб площа клітинок дорівнювала з округленням до площі зони та врахувавши, що сума отриманих клітинок за кожною областю дорівнювала їх кількості на сітці. Якщо площа логістичного парка більша за суму площ функціональних зон, то зменшимо територію планування до необхідної, а решту залишимо не зайнятою для майбутнього розвитку. Якщо навпаки, виділена територія замала для розміщення зон, то дуже ймовірно, що такий парк буде перезавантажений і не зможе ефективно виконувати свої функції.

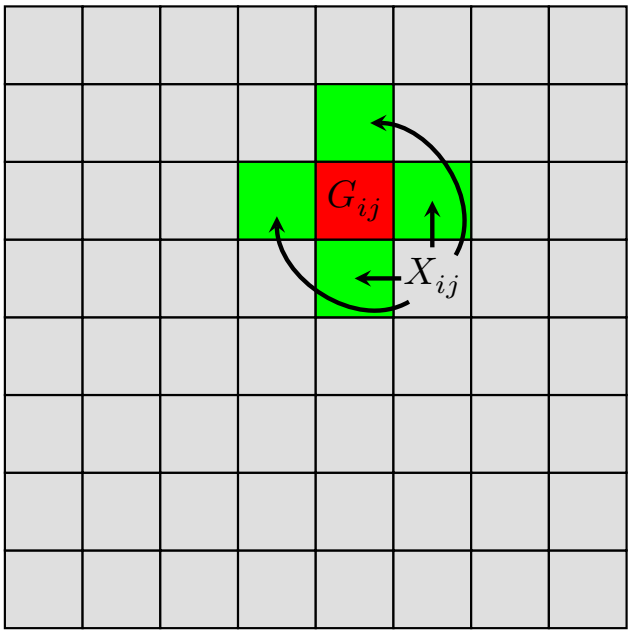



**Рисунок 1.3** – Сітка розміщення функціональних областей

Пронумеруємо від 1 для кожної клітинки сітки його координати за горизонтальною та вертикальною віссю, і нехай $n, m$ максимальний номер за горизонталлю та вертикаллю відповідно, та $G_{ij}$ буде позначати обрану функціональну область для клітинки з координатами $(i, j)$. Визначимо функцію $r\,(G_{ij},G_{i'j'})$ як кореляцію між функціональними областями $G_{ij}$ та $G_{i'j'}$ для клітинок $(i,j)$ та $(i',j')$ відповідно, яка буде позначати те,

наскільки сильно взаємопов'язані між собою ці дві області:

$$r\left(G_{ij}, G_{i'j'}\right) = \begin{cases} C_{G_{ij},G_{i'j'}} & G_{ij} \neq G_{i'j'} \\ 1 & G_{ij} = G_{i'j'} \end{cases}, \tag{1.1}$$

де $C_{G_{ij},G_{i'j'}}$ визначається як деяка константа, що менша за 1.

Введемо $R_{ij}$, що буде позначати значення суми кореляцій з усіма сусідніми клітинками:

$$R_{ij} = \sum_{(i',j') \in X_{ij}} r\left(G_{ij}, G_{i'j'}\right), \tag{1.2}$$

де $X_{ij}$ — область сусідніх клітинок до $G_{ij}$ (рис. 1.3), що визначається як

$$X_{ij} = \{(i', j') | 1 \leqslant i' \leqslant n, 1 \leqslant j' \leqslant m, |i - i'| + |j - j'| = 1\}. \tag{1.3}$$

Отже, відповідно до даних припущень, задача планування функціональних зон зводиться до розбиття клітинок сітки по різним зонам так, щоб максимізувати кореляційний ефект всього логістичного парку, тобто зробити максимальною суму кореляцій за кожною клітинкою сітки. Тому, згідно визначень, цільова функція $f$ буде визначатись як:

$$f = \max \sum_{(i,j)} R_{ij}. \tag{1.4}$$

Необхідно зауважити, що площі функціональних зон визначаються згідно пропускної здатності та функцій парку, й не змінюються під час знаходження оптимальної схеми. Тому, якщо $s_1, s_2, \ldots s_N$ — необхідні розміри площ, а $s'_1, s'_2, \ldots s'_N$ — реальні розміри на схемі, то на задачу оптимізації накладаються наступні обмеження:

$$s_1 = s'_1, s_2 = s'_2, \ldots, s_N = s'_N. \tag{1.5}$$

Щоб оцінити значення площ функціональних зон парку, можна використані формули та значення, що використані у статтях [20, 24], та

які наведені нижче.

Площа вантажної зони $S_1$:

$$S_1 = \frac{Q_1 \cdot T_1 \cdot \alpha}{\beta \cdot t}, \tag{1.6}$$

де

$Q_1$ — середньодобовий прогноз попиту на вантажі;

$T_1 = 5$ год — тривалість одного вантажного авіаперевезення;

$\alpha = 200$ м$^2$ — площа повітряного завантаження та розвантаження вантажу за один рейс;

$\beta = 0.5$ — це коефіцієнт використання площі, зазвичай береться між $20\% - 60\%$;

$t = 10$ год — час робочого дня.

Площа зони міжнародного транзиту та розподілу $S_2$:

$$S_2 = \frac{Q_2 \cdot T_2 \cdot \gamma}{\beta \cdot t}, \tag{1.7}$$

де

$Q_2$ — добовий обсяг вантажів, що вантажиться в зоні;

$T_2 = 7$ год — тривалість одного вантажного авіаперевезення;

$\gamma = 3$ м$^2$ — площа повітряного завантаження, зазвичай приймається $1.5 - 3$ м$^2$;

$\beta = 0.5$ — це коефіцієнт використання площі, зазвичай береться між $20\% - 60\%$;

$t = 10$ год — час робочого дня.

Площа митної зони $S_3$:

$$S_3 = \frac{Q_3 \cdot T_3 \cdot \delta}{\beta \cdot L_3 \cdot \epsilon \cdot X_3}, \tag{1.8}$$

де

$Q_3$ — обсяг вантажів, що надходять та вивозяться з митного оформлення;

$T_3 = 180$ днів — тривалість зберігання;

$\delta = 0.9$ м$^2$ — коефіцієнт складування, який є коефіцієнтом складування, зазвичай становить близько 80%;

$\beta = 0.5$ — це коефіцієнт використання площі, зазвичай береться між $20\% - 60\%$;

$L_3 = 3$ — кількість шарів укладання у вантажному складі, зазвичай приймається $2.5 - 4$;

$\epsilon = 0.5$ — коефіцієнт дисбалансу, який відноситься до неконтрольованих факторів у логістичному парку, зазвичай приймається $0.5 - 0.8$;

$X_3 = 2 \frac{\text{шт}}{\text{м}^2}$ — обсяг зберігання товару на одиницю площі, зазвичай приймається $1.5 - 3$ одиниць на м$^2$.

Площа зони обробки з доданою вартістю $S_4$:

$$S_4 = \frac{Q_4 \cdot T_4 \cdot \mu \cdot \delta}{W_4 \cdot t_4}, \tag{1.9}$$

де

$Q_4$ — обсяг вантажів, що надходять та вивозяться з зони;

$\mu = 0.9$ — частка товарів, які необхідно обігнути та переробити;

$\delta = 5$ — коефіцієнт пікової роботи;

$T_4 = 30$ днів — час обробки товару з одного маршруту;

$W_4 = 0.8 \frac{\text{шт}}{\text{м}^2}$ — це обсяг обробки товарів на одиницю площі, зазвичай приймається $0.8 - 1$ одиниць на м$^2$;

$t_4 = 10$ год — кчас робочого дня.

Площа зони міжнародної торгівлі $S_5$:

$$S_5 = H_5 \cdot d_5, \tag{1.10}$$

де

$H_5 = 200$ — це кількість торгових клієнтів, розміщених у зоні міжнародної торгівлі;

$d_5 = 1000$ м$^2$ — це площа, необхідна для кожного клієнта-торговця.

Площа допоміжної зони $S_6$:

$$S_6 = H_6 \cdot d_6, \tag{1.11}$$

де

$H_6 = 10000$ — це середня кількість персоналу в зоні допоміжного обслуговування;

$d_6 = 20$ м$^2$ — це площа зони допоміжного обслуговування, зайнята персоналом на одну особу.

## 1.5 Постановка завдання на дипломну роботу

Як зазначається в статті [4], дослідники досить обмежено застосовували генетичний алгоритм та його варіації в задачах авіаційного планування, і в роботі на пробному плануванні логістичного парку аеропорта була реалізована вдосконалена версія генетичного алгоритму. Нова версія алгоритму показала покращений результат за різними параметрами, і на основі цього робиться висновок про те, що цей напрямок є перспективним, однак необхідний подальший аналіз.

Тому завданням дипломної роботи ставиться є програмна реалізація раніше описаної математичної моделі оптимальної схеми розміщення авіаційного логістичного парку за допомогою класичного генетичного алгоритму та його кількох модифікацій. На основі отриманих результатів провести порівняння ефективності застосувань використаних алгоритмів, та зробити висновки про перспективи застосування модифікацій генетичного алгоритму в плануванні функціональних зон авіаційного логістичного парку.

## Висновки до розділу 1

У розділі було дане пояснення, що таке сучасний логістичний парк, які бувають його варіації за різними критеріями, та які позитивні ефекти надає компаніям. Висвітлено, чим відрізняється авіаційний логістичний парк від інших та які ключові зони в собі містить. Була надана теоретична оцінка площ функціональних областей логістичного парка аеропорту і була описана оптимізаційна модель планування цих зон на основі рівномірної сітки.

Загалом логістичний парк є складним та комплексним об'єктом, який має також враховувати попит різних галузей та їх розташування, і тому розробка ефективних планів парку та вдосконалення алгоритмів під задачу планування функціональних зон є складною і водночас важливою задачею у сфері логістичних поставок.

# 2 АНАЛІЗ СУЧАСНИХ МЕТОДІВ Й АЛГОРИТМІВ ВИРІШЕННЯ ПРОБЛЕМИ ПОШУКУ ГЛОБАЛЬНИХ ЕКСТРЕМУМІВ ФУНКЦІЙ / ФУНКЦІОНАЛІВ З ОБМЕЖЕННЯМИ

У розділі описуються загальні відомості, опис роботи, можливі модифікації та сфери застосування генетичного оптимізатора, оптимізатора на основі рою частинок, а також надаються загальні відомості про комбінації детермінованих та стохастичних оптимізаторів.

## 2.1 Генетичний оптимізатор

Як відомо, у природі більш здорові організми мають більшу ймовірність вижити. Це допомагає їм передати свої «гарні» гени наступному поколінню. З часом ці гени, що дозволяли виду краще виживати, поступово поширюються на всю популяцію та стають домінуючими в наступних поколіннях. Надихаючись цією концепцією виживання найбільш пристосованих істот, для обчислень був розроблений генетичний оптимізатор [16].

Отже, генетичний алгоритм (англ. Genetic Algorithm, GA) — це алгоритм оптимізації, мотивований процесом еволюції, і належить до великого класу еволюційних алгоритмів в інформатиці та обчислювальній математиці. Цей оптимізатор часто використовують для створення високоякісних рішень для оптимізації та пошуку рішень проблем, зосереджуючись на операторах з області біології, таких як відбір, схрещування або мутація [1].

GA відноситься до еволюційних алгоритмів. Еволюційні алгоритми використовуються для вирішення тих задач, які наразі не мають чітко визначеного ефективного рішення. Цей підхід використовується для

вирішення задач оптимізації (планування, найкоротший шлях тощо), а також у моделюванні та імітації, де використовуються функції випадковості [1].

Для реалізації GA використано ряд адаптованих термінів з біології, серед яких найбільше використовують [2]:

– Хромосома або індивід — можливе рішення поставленої задачі. Кожна хромосома складається з фіксованого набору генів (рис. 2.1).

– Ген — це одна позиція елемента хромосоми, яка є бітом або числом.

– Популяція — набір усіх хромосом (рис. 2.1).

– Придатність — значення цільовії функції (функціоналу) з умов задачі для певного індивіду. Більш пристосованим індивідумом важається той, хто має краще значення цільовії функції.

– Селекція або відбір — це процес відбору хромосом з поточної популяції, які стануть батьками для нових хромосом.

– Схрещування — це процес утворення нових індивідуумів шляхом певного комбінування генів з хромосом, обраних під час селекції.

– Мутація — це випадкова деформація деяких генів хромосоми.

– Заміщення поколінь — це відбір популяції до наступного покоління. Базовий метод відбирає найбільш придатних особин з поточної популяції та з утворених нащадків.

– Критерії зупинки — це критерій, при виконанні якого оптимальне рішення вважається знайденим та алгоритм зупиняє роботу. Зазвичай за критерій зупинки приймається досягнення максимальної кількості поколінь або досягнення одним з індивідуумів значення придатності, що є нижчим/вищим за порогове значення.

Опис класичного генетичного оптимізатора наведений на блок-схемі (рис. 2.2) [1].

В GA використовуються різноманітні схеми створення та відбору хромосом під час процесу роботи оптимізатора, а саме метод кодування генів в хромосомах, схрещення, мутацію та відбір. Розглянемо кожну групу.

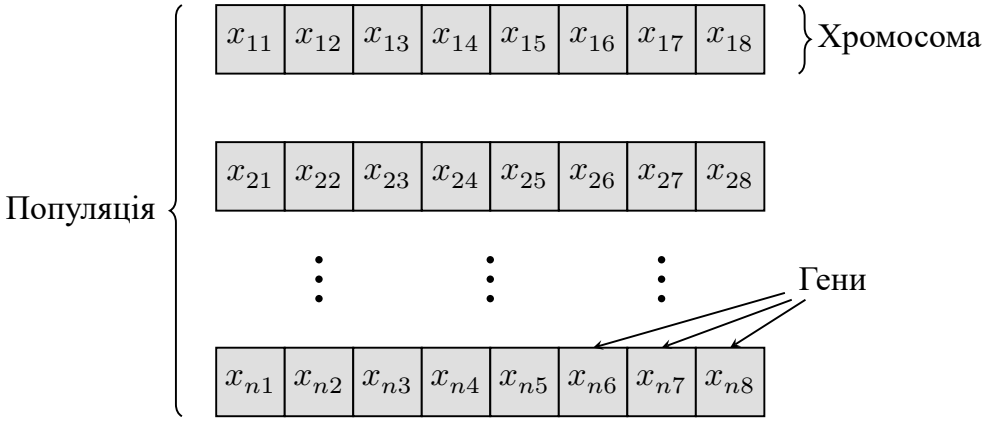



**Рисунок 2.1** – Представлення хромосом в генетичному оптимізаторі

Для більшості обчислювальних задач важливу роль відіграє схема кодування генів, які розрізняються відповідно до предметної області. Найбільш відомі схеми кодування: двійкова, вісімкова, шістнадцяткова, перестановка, на основі значень і дерево [11].

Двійкове кодування є загальновживаною схемою кодування. Кожен ген представлений у вигляді значення 1 або 0, і в такій формі хромосома забезпечує швидшу реалізацію схрещування та мутації, однак для перетворення в двійкову форму потрібні додаткові зусилля, а точність алгоритму залежить від двійкового перетворення. У вісімковій та шістнадцятковій схемі кодування хромосома представлена у вигляді ряду чисел в відповідній системі числення, та такі схеми зазвичай використовуються в задачах упорядкування. У схемі кодування на основі значень ген представлений за допомогою деякого дійсного або цілого чисел або символа, і в основному використовується в нейронних мережах для знаходження оптимальних ваг. У кодуванні у вигляді дерева ген представлений деревом функцій або команд й використовується в еволюційних програмах або виразах [11].

Відбір є важливим кроком у генетичних алгоритмах, який визначає, чи братиме певний індивідуум участь у процесі відтворення, чи ні. Відомими методами відбору є з колесом рулетки, ранговий, турнірний, елітарний та стохастичний універсальний, але також застосовують усічений, стаціонарний, локальний, нечіткий, відбір Больцмана та

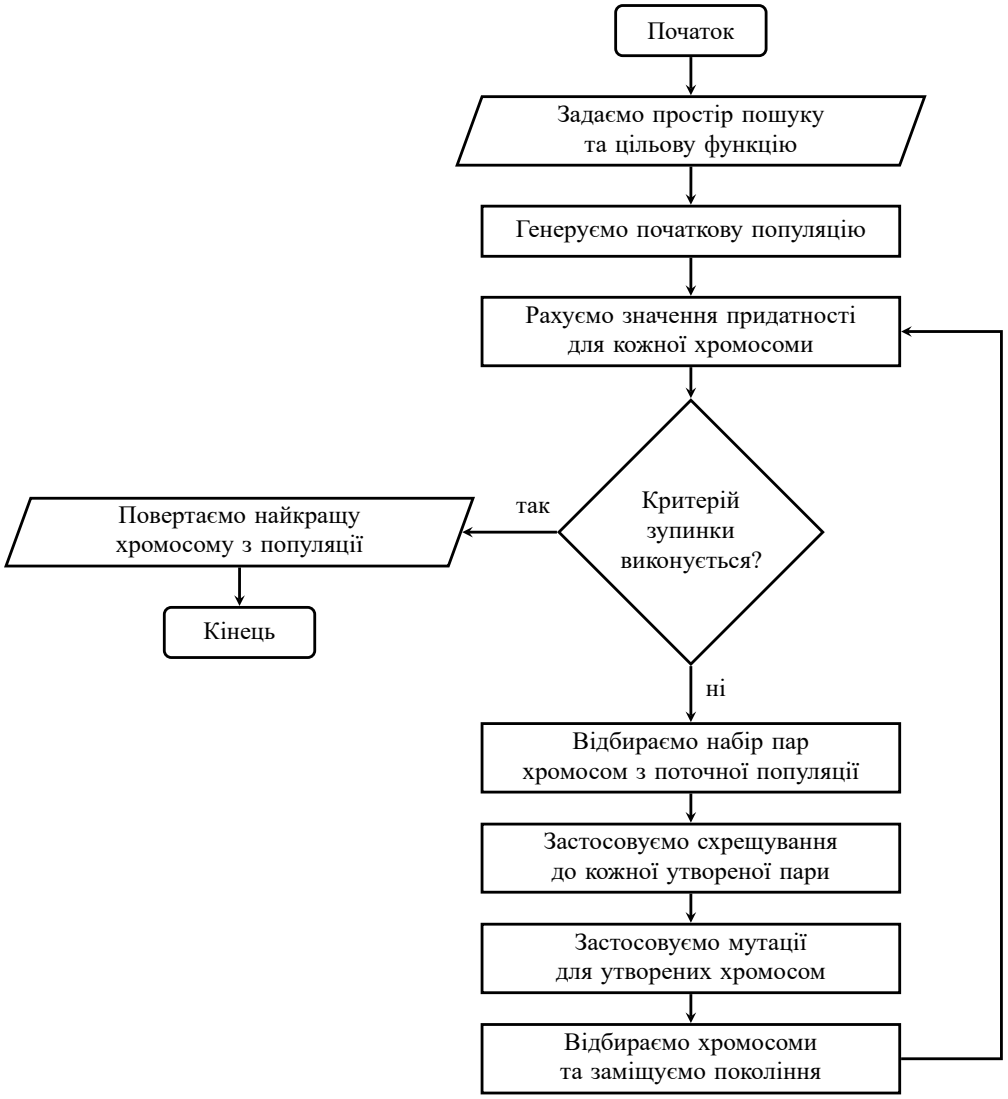



**Рисунок 2.2** – Блок-схема роботи генетичного оптимізатора

інші [16, 11].

Розглянемо деякі з них. Нехай $M$ — кількість необхідних батьків, тоді [2, 11]:

– при відборі колесом рулетки кожній хромосомі надається значення ймовірності вибору, яке пропорційне її значенню придатності, і далі хромосоми відбираються випадковим чином відповідно до наданих ймовірностей, поки не набереться $M$ хромосом;

– відбір за рангом аналогічний до відбору колесом рулетки, де замість значення придатності використовуються ранги, які надаються їм відповідно до рівня фізичної підготовки;

– в турнірному відборі беруть випадковим чином $s$ особин із

популяції, де зазвичай встановлюють $s = 2$, а потім відбирають найпридатнішу з цих $s$ особин, і це повторюється, поки не набереться $M$ хромосом;

– в стохастичному універсальному відборі хромосоми обираються випадковим чином, де для кожної встановлена однакова ймовірність відбору, поки не набереться $M$ хромосом;

– в елітарному відборі обираються $M$ найбільш придатних особин.

Схрещування в алгоритмі використовується для створення нащадків шляхом поєднання генетичної інформації двох (або більше) батьків. До відомих методів схрещування належить одноточкове, двоточкове, k-точкове, рівномірне, частково узгоджене, порядкове, із збереженням пріоритету, напіврівномірне, циклічне, з трьома батьками та інші. Розглянемо для прикладу деякі найпопулярніші методи [16, 2, 11]:

– У одноточковому схрещувані випадковим чином обирається точка перетину, до якої гени батьків залишаються на місці, а після точки хвости міняють місцями, і таким чином утворені дві хромосоми будуть являти собою нових нащадків в популяції.

– В двоточковому схрещувані вибираються дві випадкові точки перетину, і лише між точками гени міняються місцями.

– K-точкове схрещування є узагальненням двох попередніх, де гени, що міняються місцями, йдуть почергово з тими, що залишаються на місці.

– В рівномірному схрещуванні кожен ген або успадковується від першого батька з деякою фіксованою ймовірністю, або інакше береться від другого батька.

– В порядковому схрещуванні використовується випадковий двійковий шаблон для створення дітей, в якому якщо в деякому місці стоїть 1, то відповідний ген успадковується від першого батька, а якщо стоїть 0, то від другого батька.

Головною проблемою схрещування полягає у відсутності введення різноманітності хромосом. Якщо всі хромосоми стали неякісними (потрапили у локально оптимальні рішення), схрещування не призводить

до різних рішень із новими генами, що відрізняються від генів батьків. Щоб вирішити це, в алгоритмі GA впроваджено та активно використовується механізм мутації генів [16].

Основна ідея мутації полягає в випадковій зміні генів в одному чи багатьох місцях. Кількість мутацій контролюється ймовірністю мутації $P_m$, яка зазвичай зберігається якомога нижчою, щоб уникнути поведінки алгоритму як випадкового пошуку. Є багато методів мутацій, однак їх реалізація залежить від представлення генів: наприклад, у двійковій схемі до генів можна застосовувати інверсію, а у дійсному представленні можна застосувати заміну гена на значення з певного діапазону. До використовуваних методів мутації належать одноточкова, з порозрядною інверсією, випадкової змінної, гранична, рівномірна, нерівномірна, з різними ймовірностями, зсув та інші [16, 2, 11].

Після створення нового потомства готується нова популяція для наступного покоління. В одному з найпростіших випадків майбутня популяція буде складатись з нових нащадків та батьків, яких не допустили до схрещування. Крім такого методу, використовують також інші, а саме [2]:

– Метод видалення всіх замінює все поточне покоління новими нащадками.

– Метод заміни в стаціонарному стані замінює невелику кількість хромосом популяції на таку саму кількість нових індивідуумів.

– Метод елітарного заміщення вибирає найпридатніші хромосоми з поточної популяції та з нових нащадків.

– Метод видалення $n$-останніх замінює найгірші $n$ хромосоми з поточної популяції на $n$ хромосом з нових нащадків.

Окрім варіантів GA, які можна отримати, комбінуючи різні схеми створення та відбору хромосом, також до основних категорій модифікацій можна віднести багатооб'єктні, паралельні, хаотичні та гібридні GA. В багатокритеріальному ГА основною метою є створення оптимального фронту Парето таким чином, щоб не було подальшого покращення

будь-якої функції пристосованості без порушення інших функцій пристосованості. Основною метою паралельних GA полягає в тому, щоб покращити обчислювальний час і якість рішень через розподілення обчислень між різними обчислювальними процесорами. Хаотичні GA намагаються усунути проблему передчасної збіжності, що часто буває в класичному GA, шляхом включення різних хаотичних систем. Також застосовуються гібридні GA, комбінуючи класичний GA з іншими методами оптимізації, для того, щоб отримати кращу якість рішень, кращу ефективність, гарантію отримання рішень чи знаходження оптимізованих параметрів [11].

Вчені використовують генетичний алгоритм для вирішення проблем оптимізації в багатьох областях, таких як обробка зображень, обробка відео, медична візуалізація, розробка ігор, планування об'єктів, проектування мережі постачання, розробка планів, прогнозування, контроль запасів, кластеризація, розробка програмного забезпечення, обробка природної мови, розробка системи рекомендацій, балансування навантаження, локалізація, розподіл смуги пропускання та призначення каналів [1, 2].

## 2.2 Оптимізатор на основі рою частинок

Оптимізатор на основі рою частинок (англ. Particle Swarm Optimization, PSO) — це стохастичний алгоритм оптимізації, що належить до групи алгоритмів ройового інтелекту, та який використовує концепції соціальної поведінки тварин, яку можна спостерігати у зграї птахів чи риб. Цей алгоритм моделює соціальну поведінку птахів у зграї, де завдяки поєднанню особистого та соціального досвіду вони наближаються до своєї мети — їжі. Вони постійно оновлюють свою позицію відповідно до власної найкращої позиції та найкращої позиції всієї зграї та перегруповуються, створюючи оптимальну форму [10, 7].

Відповідно до цієї концепції, зграя птахів інтерпретується як рій

частинок деякої розмірності $n$, де кожна частинка рою в деякий момент $t$ своїм розташуванням $X_i^t = (x_{i1}^t, x_{i2}^t, ..., x_{iN}^t)$, $i$ = 1,2,...,$n$ представляє можливий варіант рішення задачі, а $N$ — розмірність простору рішень задачі. Замість оцінки відстані до їжі буде використовуватись цільова функція $f$, і головною метою оптимізатора буде знаходження найкращого рішення [18].

Окрім знання свого положення, кожна $i$-та частинка має власну швидкість $V_i^t = (v_{i1}^t, v_{i2}^t, ..., v_{iN}^t)$, а також їй відоме особисте найкраще значення $P_{\text{best } i}^t = (p_{\text{best } i1}^t, p_{\text{best } i2}^t, ..., p_{\text{best } iN}^t)$ — найкраща позиція, отримана на момент $t$ самою частинкою, та глобальне найкраще значення $G_{\text{best}}^t = (g_{\text{best } 1}^t, g_{\text{best } 2}^t, ..., g_{\text{best } N}^t)$ — найкраща позиція будь-якої частинки, отримана на момент $t$ у всій сукупності під час процесу пошуку в просторі рішень [18]:

$$f(P_{\text{best } i}^t) = \max_{k=1,2,...,t} f(X_i^k), \tag{2.1}$$

$$f(G_{\text{best}}^t) = \max_{\substack{k=1,2,...,t \\ i=1,2,...,n}} f(X_i^k) = \max_{i=1,2,...,n} f(P_{\text{best } i}^t). \tag{2.2}$$

У PSO регулювання швидкості розглядається як головна функція, оскільки це основний механізм, який використовується для переміщення положення частинки для пошуку оптимального рішення в просторі пошуку. Для $i$-тої частинки кожна координата $V_i^t = (v_{i1}^t, v_{i2}^t, ..., v_{iN}^t)$ в $(t+1)$-й ітерації оновлюється наступним чином [10, 7]:

$$v_{ij}^{t+1} = \omega \cdot v_{ij}^t + c_1 r_1 \cdot (p_{\text{best } ij}^t - x_j^t) + c_2 r_2 \cdot (g_{\text{best } j}^t - x_j^t),\ j = 1,...,N\ , \tag{2.3}$$

де

$\omega$ — додатний коефіцієнтом, яку ще називають «вагою інерції»;

$r_1, r_2$ — випадкові величини, що рівномірно розподілені на $[0, 1]$;

$c_1, c_2$ — додатні коефіцієнти, які ще називають «коефіцієнтами прискорення».

Кожна компонента рівняння 2.3 має свою інтерпретацію, а саме

(рис. 2.3) [10, 7]:

– Імпульсна частина $\omega \cdot v_{ij}^t$ — слугує пам'яттю про минулий політ, оскільки використовує попередню швидкість. Вона також вважається інерційною компонентою, який забезпечує баланс між дослідженням і використанням кожної частинки в просторі пошуку.

– Когнітивна частина $c_1 r_1 \cdot (p_{\text{best } ij}^t - x_j^t)$ — частина, яка приводить частинку до її найкращого положення та еквівалентна відстані частинки від її особистого найкращого положення до поточного моменту.

– Соціальна частина $c_2 r_2 \cdot (g_{\text{best } j}^t - x_j^t)$ — складова, яка приводить частинку до найкращого положення, визначеного роєм.

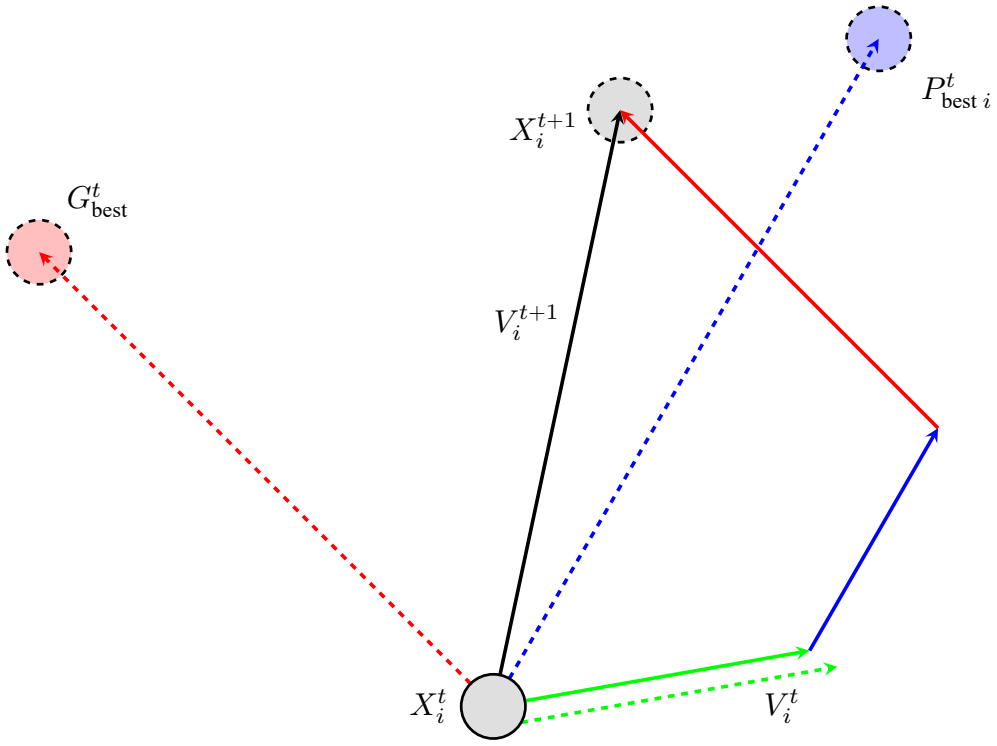



**Рисунок 2.3** – Зміна швидкості та положення в оптимізаторі на основі рою частинок

Для вектора швидкості зазвичай встановлюється верхня межа $V_{\max}$, щоб уникнути вибуху швидкості та розбіжності, однак знайти правильне значення $V_{\max}$ доволі важко. Для великих значень $V_{\max}$ частинки можуть летіти майже випадковим чином і пропустити оптимальне рішення. І

навпаки, для малих значень $V_{\max}$ частинки матимуть дуже вузький простір пошуку, що може призвести до потрапляння в локальний оптимум. Для вирішення цієї проблеми, часто максимальна швидкість $V_{\max}$ встановлюється наступним чином [10, 18]:

$$V_{\max} = \delta \cdot (X_{\max} - X_{\min}), \tag{2.4}$$

де $X_{\max}$, $X_{\min}$ є максимальним та мінімальним значенням меж простору пошуку відповідно, і $\delta$ є деяким коефіцієнтом з діапазону $(0, 1]$.

Після оновлення швидкості, положення $X_i^t = (x_{i1}^t, x_{i2}^t, ..., x_{iN}^t)$ змінюється відповідно до наступного рівняння [10]:

$$x_{ij}^{t+1} = x_{ij}^t + v_{ij}^{t+1},\ j = 1,...,N. \tag{2.5}$$

Як правило, використовують два типи критеріїв зупинки для завершення циклу PSO. У першому критерії виконання PSO припиняється, коли досягається попередньо визначена кількість ітерацій. Другим критерієм є кількість оцінок функції, яка розраховується як добуток розміру рою на максимальну кількість ітерацій [18].

Опис алгоритму роботи класичного PSO наведений на блок-схемі (рис. 2.4) [7].

Загалом PSO має три основні контрольні параметри, регулювання яких може значно підвищити продуктивність: вага інерції $\omega$ та коефіцієнти прискорення $c_1$, $c_2$. Для інерційної ваги механізми налаштування можна розділити на три види, а саме [18]:

– $\omega$ — константа або задається через випадкову величину;

– $\omega$ змінюється з часом;

– $\omega$ є адаптивною величиною, яка регулює своє значення на основі параметра зворотного зв'язку.

Для $c_1$, $c_2$ рекомендованою та найпоширенішою стратегією є встановлення їх значення рівним 2, а також розповсюдженим є значення 1,49. Але окрім визначення їх як константи, пропонується стратегія

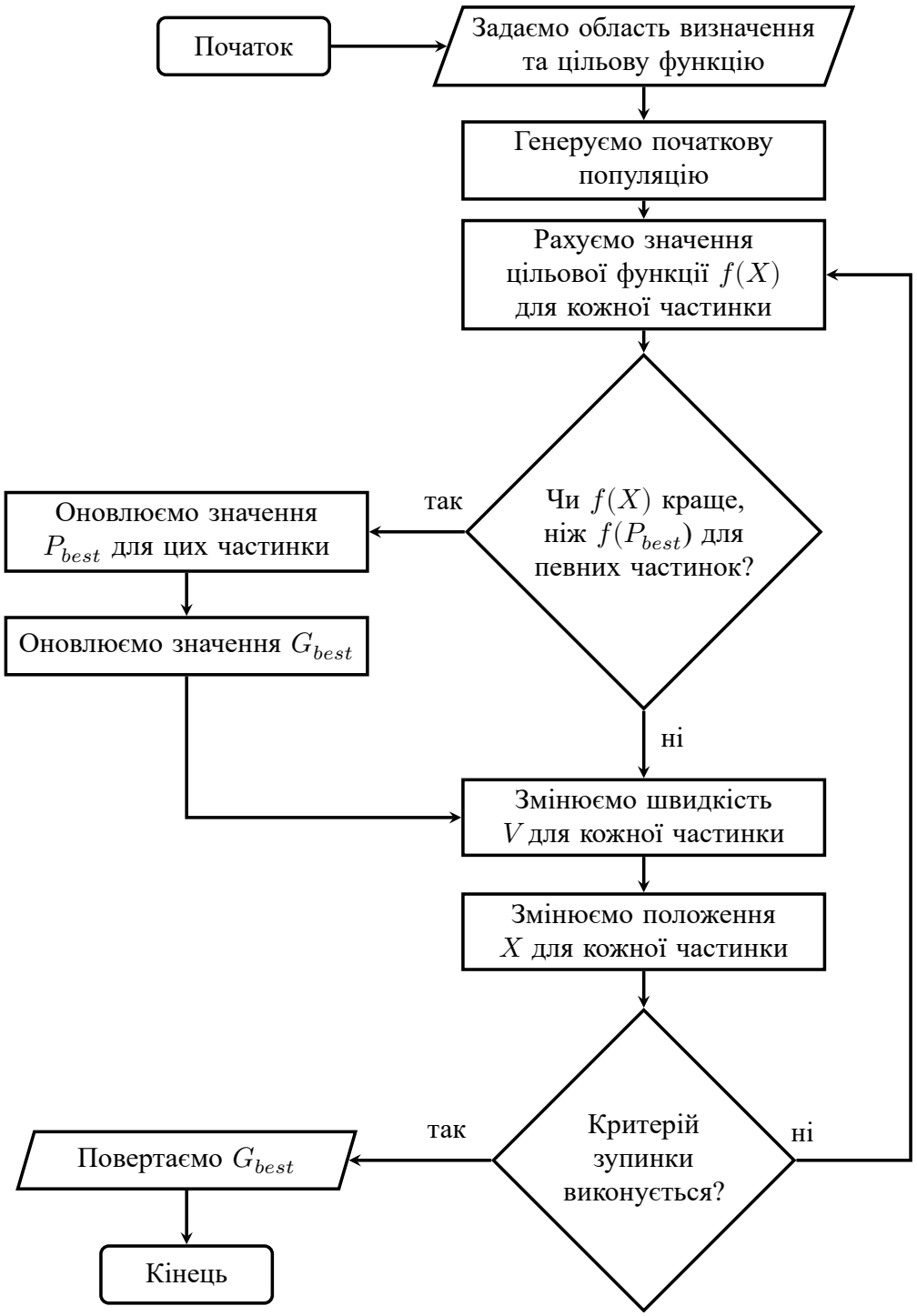



**Рисунок 2.4** – Блок-схема роботи оптимізатора на основі рою частинок

встановлювати на початку великі значення $c_1$ і мале значення $c_2$, щоб дозволити частинкам виконувати розширений пошук, й потім поступово зменшувати $c_1$ та збільшувати $c_1$, щоб частинки більше зосередилися на локальному пошуку поблизу найперспективнішого рішення [18].

Окрім підбору параметрів, для покращення продуктивності

використовуються інші топології сусідства. В класичному PSO використовується топологія зірки, де частинка має зв'язок з рештою рою і відповідно $G_{\text{best}}$ рахується на основі всієї популяції. Однак також використовуються інші типи топології, де частинка пов'язана лише з обмеженою кількістю частинок навколо неї, і $G_{\text{best}}$ рахується для кожного окремо лише на основі них. До таких топологій належить топологія кільця, топологія фон Неймана, динамічна топологія та інші [18].

Для PSO створено багато модифікацій, які можна поділити на три види: гібридизація, вдосконалення та адаптація під тип задачі. Під гібридизацією мається на увазі створення моделі, що об'єднує два або більше алгоритми, для використання їх переваг, одночасно зменшуючи їхні недоліки. В дослідженнях використовують гібридизацію PSO з метаевристичними алгоритмами, до яких належить генетичний оптимізатор, диференціальна еволюція та різні види алгоритмів на основі ройового інтелекту, з нейронними мережами, з методом опорних векторів, та інші. Для покращення окремих аспектів, як продуктивність чи збіжність, були розроблені різні вдосконалення PSO з використанням стратегії навчання, нечіткої логіки, мутацій, та інші. А для деяких задач, як бінарна оптимізація, задачі оптимізації з невизначеністю, чи багатокритеріальна оптимізація, були створені різні адаптовані версії PSO, а саме бінарний PSO, хаотичний PSO та багатокритеріальний PSO відповідно [7, 18].

Сфера практичного застосування PSO та його модифікацій є дуже широкою, і до неї належить [7]:

– охорона здоров'я, як наприклад діагностування хвороб та сегментація зображень;

– дослідження навколишнього середовища, як наприклад прогнозування температури та опадів;

– промисловість, як наприклад проєктування та розподіл навантаження системи;

– комерція, як наприклад прогнозування цін та прибутку;

– розробка розумного міста, як наприклад контроль теплового навантаження будівель та планівання роботи світлофора;

– інші практичні задачі, як наприклад планування завдань та кластеризація данних.

## 2.3 Детерміновані глобальні оптимізатори екстремальних задач

Існує два логічних способи класифікації стратегій глобальної оптимізації, а саме: детермінований та стохастичний. Метод вважається детермінованим, якщо всі кроки його алгоритму точно визначені за однакових початкових умов. Такі методи базуються на певній теорії, за допомогою якої обчислюється розв'язок задачі. Детерміновані методи шукають увесь простір можливих рішень і гарантують оптимальність та показують відмінні результати, коли простір пошуку є опуклим і неперервним. Але якщо ці умови не гарантовані, ці методи стають неефективними, надаючи неприйнятні рішення або не забезпечуючи необхідну точність. Оскільки більшість безперервних чи комбінаторних моделей є NP-складними, і навіть якщо для них можливе застосування детермінованих підходів, то вимагатиметься надзвичайно велика кількість обчислювальних ресурсів. Через це, пряме використання детермінованих оптимізаторів є непрактичним для більшості екстремальних задач [17, 3].

Наведемо деякі групи часто застосовуваних детермінованих оптимізаторів [17]:

– Методи гілок і меж. У цих методах вхідна множина розбивається на підмножини (гілки), для яких знаходяться оцінки (межі), на основі яких обираються наступні множини для повторення процесу, поки не знайдеться оптимальне рішення. Цей загальний підхід включає багато різних випадків і допускає значні узагальнення. Методи гілок і меж застосовні до різноманітних структурованих та неперервних моделей

оптимізації.

– Стратегії перерахунку. Ці методи базуються на повному переліку всіх можливих рішень. Застосовується до задач комбінаторної оптимізації та до певних структурованих неперервних моделей оптимізації.

– Гомотопічні і траєкторні методи. Ці стратегії ставлюь за мету відвідування (явне перерахування) усіх стаціонарних точок цільової функції, після чого складаються списки глобальних та локальних оптимумів. Методи застосовні до гладких задач оптимізації, але зазвичай обчислювальні вимоги є дуже значними.

– Інтегральні методи. Ці методи спрямовані на визначення супремуму цільової функції шляхом її апроксимації.

– Стратегії релаксації (зовнішнього наближення). У цьому підході оптимізаційна проблема замінюється послідовністю більш легких підпроблем. Такі методи застосовні до різноманітних структурованих моделей оптимізації.

## 2.4 Комбінація детермінованих і стохастичних оптимізаторів

У зв'язку з великими обчислювальними затратами та складністю побудови структурованої моделі, на практиці дуже часто використовують стохастичні методи оптимізації. Стохастичні оптимізатори складаються з аналізу ймовірнісних правил і використання випадкових значень у своїх процедурах. Стохастичні методи не завжди можуть точно знайти оптимальне рішення, і в деяких задачах оптимальне рішення дещо відрізняється при кожному виконанні цього алгоритму. Таким чином, методи стохастичної оптимізації використовують кількісну оцінку невизначеності для отримання рішень, які оптимізують проблему. Як правило, ці методи не гарантують точного чи високоточного рішення, однак вони зазвичай пропонують задовільний розв'язок, дуже близький

до оптимального, за більш короткий час [17].

Щоб частково вирішити згадані проблеми оптимізаторів, у різних роботах пропонуються схеми гібридних стратегій з застосуванням детермінованих та стохастичних оптимізаторів, які врівноважують велику обчислювальну затратність детерміністичних та неточність стохастичних методів. Але часто застосування детерміністичного підходу неможливе або не є доцільним, тому також дуже розповсюджені схеми поєднання лише стохастичних алгоритмів. У цій стратегії алгоритми поєднуються так, щоб підтримувався оптимальний баланс між точністю знайденого розв'язку та швидкістю збіжності гібридного методу, оскільки передчасна збіжність призводить до недостатньої точності кінцевого рішення, а повільна збіжність призводить до нижчої ефективності пошуку рішення [13, 19].

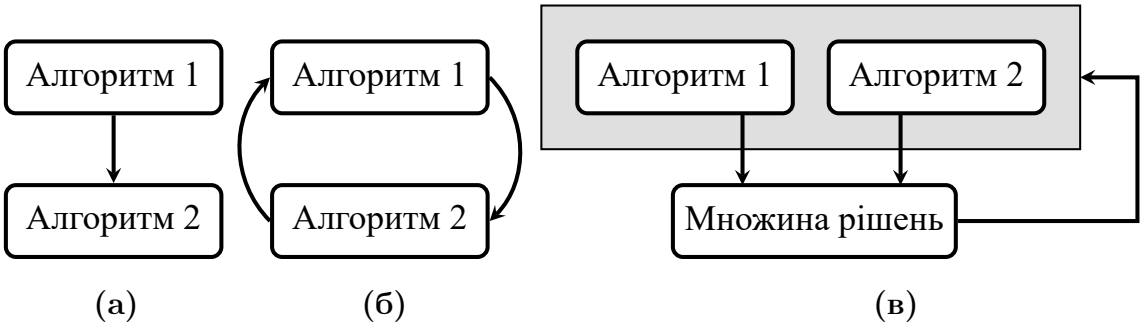



**Рисунок 2.5** – Типи комбінацій детермінованих і стохастичних оптимізаторів: (а) послідовно, (б) по черзі, (в) паралельно

Ці комбінації детермінованих і стохастичних оптимізаторів розділити на три групи, а саме [13, 19]:

– використання цих методів послідовно, де спочатку використовується стохастична оптимізація, а знайдені рішення далі чи вдосконалюється за допомогою методу детермінованої оптимізації, чи здійснюється навколо них локальний пошук за допомогою іншого методу стохастичної оптимізації (рис. 2.5);

– використання стохастичні та детерміновані стратегій по черзі у

циклічний спосіб (рис. 2.5);

– використання цих методів паралельно один до одного, однак при цьому вони можуть обмінюватись інформацією, наприклад про знайдені рішення (рис. 2.5).

## Висновки до розділу 2

У розділі було розглянуто два стохастичних популяційних методи, а саме генетичний алгоритм (GA) та метод на основі рою частинок (PSO), які широко використовуються в оптимізаційних задачах для пошуку оптимальних чи субоптимальних розв'язків у просторі рішень.

Для більш ефективної роботи методів та алгоритмів важливою складовою є підбір оптимальних параметрів роботи, від яких суттєво залежить час роботи та якість знайдених розв'язків, що властиво для обох методів, однак для PSO це можна простіше зробити завдяки меншій кількості параметрів. Окрім цього, було розроблено багато модифікацій та адаптацій під різні типи задач, як задачі з багатьма цілями чи з невизначеністю, та для досягнення кращих результатів, і в цьому аспекті завдяки можливості використання різноманітних схем кодування та операторів схрещування, мутації та селекції, GA є дуже гнучким та універсальним методом для задач оптимізації.

# 3 РЕАЛІЗАЦІЯ МОДЕЛІ ПОБУДОВИ ОПТИМАЛЬНОЇ СХЕМИ РОЗМІЩЕННЯ ЛОГІСТИЧНОГО ПАРКУ АЕРОПОРТУ

У розділі наводяться обрані параметри для моделі, які модифікації обрано для розробки, результати тестування та аналіз відповідних програмних розробок моделей.

## 3.1 Структура прикладного програмного комплексу розв'язання основної задачі оптимізації

Для дослідження та тестування для схеми розміщення функціональних зон були використані сітки розміром $5 \times 5$ та $3 \times 9$ на основі математичної моделі, що була описана у першому розділі. Для зручності обчислень замість назв функціональних зон використаємо їх кодування у вигляді числа: «0» — вантажна зона, «1» — митна зона, «2» — зона транзиту, «3» — зона обробки з доданою вартістю, «4» — зона міжнародної торгівлі, «5» — допоміжна зона, «6» — зона інформаційного обслуговування. Оскільки значень площ функціональних зон та кореляцій між ними не є визначеними, тому були використані вже знайдені їх оцінки зі статті [20], тому:

1) вантажна зона $s_1 = 1{,}28$ км$^2$;
2) митна зона $s_2 = 0{,}54$ км$^2$;
3) зона транзиту $s_3 = 0{,}3$ км$^2$;
4) зона обробки з доданою вартістю $s_4 = 0{,}73$ км$^2$;
5) зона міжнародної торгівлі $s_5 = 0{,}2$ км$^2$;
6) допоміжна зона $s_6 = 0{,}2$ км$^2$;
7) зона інформаційного обслуговування $s_6 = 0{,}06$ км$^2$.

Матриця кореляцій зон:

$$C = \begin{pmatrix} 1 & 0{,}4 & 0{,}3 & 0{,}2 & 0{,}2 & 0 & 0 \\ 0{,}4 & 1 & 0{,}3 & 0{,}1 & 0{,}2 & 0 & 0 \\ 0{,}3 & 0{,}3 & 1 & 0{,}1 & 0{,}1 & 0 & 0{,}1 \\ 0{,}2 & 0{,}1 & 0{,}1 & 1 & 0 & 0 & 0{,}1 \\ 0{,}2 & 0{,}2 & 0{,}1 & 0 & 1 & 0{,}1 & 0 \\ 0 & 0 & 0 & 0 & 0{,}1 & 1 & -0{,}1 \\ 0 & 0 & 0{,}1 & 0{,}1 & 0 & -0{,}1 & 1 \end{pmatrix}.$$

Маючи інформацію про розміри площ, можна оцінити, скільки місця буде займати кожна функціональна зона в математичній моделі. Тоді для сітки розміром $5 \times 5$:

1) вантажна зона — 9 клітинок;

2) митна зона — 4 клітинки;

3) зона транзиту — 2 клітинки;

4) зона обробки з доданою вартістю — 5 клітинок;

5) зона міжнародної торгівлі — 2 клітинки;

6) допоміжна зона — 2 клітинки.

7) зона інформаційного обслуговування — 1 клітинка.

А для сітки розміром $3 \times 9$:

1) вантажна зона — 10 клітинок;

2) митна зона — 4 клітинки;

3) зона транзиту — 2 клітинки;

4) зона обробки з доданою вартістю — 6 клітинок;

5) зона міжнародної торгівлі — 2 клітинки;

6) допоміжна зона — 2 клітинки.

7) зона інформаційного обслуговування — 1 клітинка.

Розмір сітки не дуже великий, тому розрахунок кількості клітинок на кожну зону є приблизним. Щоб пололати це, можна збільшити розмірність схеми, однак це також призведе до суттевого збільшення обчислень та часу роботи для розрахунку оптимальної схеми.

## 3.2 Адаптація генетичного оптимізатора до поставленої задачі

Для реалізації в якості базового оптимізатора було обрано класичний генетичний оптимізатор (рис. 2.2) з наступними параметрами та операторами:

– схемою кодування є цілочисельною, де будь-який ген буде приймати одне значення з множини $\{0,1,2,3,4,5,6\}$, що відповідає за певну функціональну зону;

– схрещення є двоточковим, де обираються випадковим чином двві позиції, і лише гени між ними обмінюються місцями, а решта батьківських генів залишається на місці;

– мутація відбувається через перестановку певної кількості випадковим чином обраних генів хромосоми;

– під час селекції серед набору хромосом обираються лише найпридатніші.

Оскільки під час роботи генетичного оптимізатора розміри площ в нових утворених рішеннях можуть відрізнятись від необхідних, то для оптимізації буде розглядатись цільова функція, на яку накладається додатковий штраф, що обмежує відбір таких рішень до наступного покоління:

$$f_{\text{new}} = f - \lambda \cdot \sum_{k=1}^{N} \left(s_k - {s_k}'\right)^2 . \tag{3.1}$$

де

– $\lambda > 0$ — коефіцієнт, що визначає розмір штрафу;

– $f$ — початкова цільова функція (1.4);

– $s_1, s_2, \ldots s_N$ — необхідні розміри площ функціональних зон;

– $s'_1, s'_2, \ldots s'_N$ — реальні розміри на схемі.

В якості початкової популяції випадковим чином створюються хромосоми вже з правильними розмірами площ парку шляхом перестановки випадково створеного рішення. В якості критерію зупинки буде досягнення максимальної кількості ітерацій, а саме 400.

## 3.3 Адаптація оптимізатора на основі рою частинок до поставленої задачі

Для порівняння з генетичним алгоритмом та його модифікаціями також було обрано класичний оптимізатор на основі рою частинок (рис. 2.2). Оскільки метод працює в неперервному просторі, тому буде розглядатись простір $[0,n-1]^S$, де $S$ — кількість клітинок схеми, $n = 7$ — кількість функціональних зон, і номер зони $G_k$ в $k$-й клітинці для рівняння (1.2) буде визначатись через округлення відповідної координати $x_k$ неперервного простору.

Так само як в генетичному алгоритмі, під час роботи методу на основі рою частинок розміри площ в нових утворених рішеннях можуть відрізнятись від необхідних, то для таких розв'язків на цільову функцію накладається додатковий штраф (3.1).

Так само. як в GA, d якості початкової популяції випадковим чином створюються хромосоми вже з правильними розмірами площ парку, а в якості критерію зупинки буде досягнення максимальної кількості ітерацій, а саме 400.

## 3.4 Модифікації класичних оптимізаторів

*Модифікація відбору рішень для GA та PSO.* Класичні популяційні методи та алгоритми не забезпечують збереження різних обмежень, і через це часто для їх врахування накладаються штрафи на цільову функцію. Тому для задачі планування парка часто розміри площ для різних зон будуть відрізнятись від необхідних, через що частина рішень під час пошуку будуть мати слабкі значення цільової функції та відкинуться як неприйнятні. Щоб прискорити збіжність та не втрачати потенційні розв'язки, такі рішення будуть доповнюватись до нормальних наступним чином:

1) шукаємо, які зони в надлишку та яким не вистачає;

2) вибираємо випадковим чином деякі позиції надлишкових зон у кількостях, щоб повернути кількісті зон до необхідних;

3) розставляємо у ці позиції випадковим чином зони, яких не вистачає, у таких кількостях, щоб рішення знов стало прийнятним.

*Модифікація схеми та методу схрещування GA.* Ця модифікація нагадує попередню, однак схема буде представлена зворотнім чином: замість вибору, в яку клітинку потрапить яка функціональна зона, області зон будуть вже певним чином в пам'яті розташовані і в них будуть обиратись номери позицій клітинок $0, 1, \ldots, n \cdot m$, де $(n,m)$ — розмірність сітки. У такій схемі обмеження на площі завжди виконується, і лише необхідно, щоб номери позицій зустрічались рівно один раз. Тому для генетичного алгоритму необхідно додатково реалізувати прибирання дублікатів і на утворені місця випадковим чином розташування втрачених номерів.

## 3.5 Порівняльний аналіз роботи класичних й модифікованих оптимізаторів

Методи, що розглядаються у роботі є стохастичними, через що отримана схема розміщення функціональних зон парку не обов'язково буде найбільш оптимальною, а значення цільової функції та час роботи алгоритмів можуть коливатись в певному проміжку. Тому, щоб усереднити результат по досягненому значенню та по часу, тестування повторювалось 100 разів для кожного методу при однакових параметрах популяції. Кінцеві результати наведені в таблицях (табл. 3.1, 3.2), усереднені графіки зміни знайденого найкращого значення цільової функції по ітераціях (рис. 3.1, 3.2) та знайдені оптимальні схеми розміщення функціональних зон авіаційних логістичних парків (рис. 3.3) наведені на рисунках. Модель та методи були реалізовані за допомогою мови програмування Python, та їх написані тести наведені у додатку до роботи, а саме тестування проводилось у середовищі Google Colab на базі Intel(R) Xeon(R) CPU @ 2.20GHz Family 6 Model 79.

**Таблиця 3.1** – Таблиця результатів оптимізації схеми розміщення авіаційного логістичного парку для схеми розміром $3 \times 9$

| Алгоритм | Розмір популяції | Кількість досягнень максимуму | Середнє | Медіана | Середній час роботи, с | Макс. | Мін. |
|---|---|---|---|---|---|---|---|
| Класичний PSO | 300 | 0 | 40,68 | 40,0 | 4,05 | 50,0 | 36,6 |
| PSO з іншим відбором | | 0 | 57,08 | 57,2 | 18,05 | 60,0 | 52,0 |
| Класичний GA | | 0 | 56,77 | 57,0 | 5,65 | 59,6 | 52,2 |
| GA з іншим відбором | | 3 | 58,99 | 59,2 | 7,03 | 60,2 | 56,4 |
| GA з іншою схемою | | 2 | 58,92 | 59,2 | 13,37 | 60,2 | 55,6 |
| Класичний PSO | 600 | 0 | 42,81 | 42,5 | 6,68 | 49,4 | 36,6 |
| PSO з іншим відбором | | 0 | 57,80 | 57,8 | 32,61 | 59,8 | 53,0 |
| Класичний GA | | 1 | 57,42 | 57,8 | 10,39 | 60,2 | 52,8 |
| GA з іншим відбором | | 3 | 59,29 | 59,4 | 13,21 | 60,2 | 56,0 |
| GA з іншою схемою | | 5 | 59,34 | 59,4 | 24,36 | 60,2 | 57,2 |

**Таблиця 3.2** – Таблиця результатів оптимізації схеми розміщення авіаційного логістичного парку для схеми розміром $5 \times 5$

| Алгоритм | Розмір популяції | Кількість досягнень максимуму | Середнє | Медіана | Середній час роботи, с | Макс. | Мін. |
|---|---|---|---|---|---|---|---|
| Класичний PSO | 300 | 0 | 39,51 | 38,8 | 4,17 | 47,2 | 34,6 |
| PSO з іншим відбором | | 1 | 51,59 | 51,6 | 17,96 | 54,2 | 49,0 |
| Класичний GA | | 1 | 52,01 | 52,0 | 5,59 | 54,2 | 49,4 |
| GA з іншим відбором | | 4 | 52,70 | 52,4 | 6,93 | 54,2 | 51,0 |
| GA з іншою схемою | | 8 | 53,01 | 53,2 | 12,73 | 54,2 | 51,0 |
| Класичний PSO | 600 | 0 | 41,14 | 40,8 | 6,79 | 48,0 | 35,2 |
| PSO з іншим відбором | | 12 | 52,31 | 52,0 | 32,15 | 54,2 | 49,8 |
| Класичний GA | | 1 | 52,11 | 52,0 | 10,48 | 54,2 | 49,6 |
| GA з іншим відбором | | 6 | 52,92 | 53,0 | 13,49 | 54,2 | 51,0 |
| GA з іншою схемою | | 22 | 53,36 | 53,4 | 23,66 | 54,2 | 51,8 |

Порівнюючи результати між собою (табл. 3.1, 3.2), в залежності від схеми, найкращий результат показав GA з модифікованою схемою та GA з модифікованим відбором, як за середнім результатом, медіаною, так і за кількістю запусків, що знайшли глобальний максимум. Окремо порівнюючи GA з модифікованою схемою та GA з модифікованим відбором, то перший має перевагу по витраченому часу приблизно в 2 рази, однак другий має кращі результати по мінімуму зі 100 запусків для двох схем.

Найгірші показники, окрім часу, має класичний PSO, оскільки цільова функція в цій задачі є складною та негладкою, та замість дослідження лише розв'язків, для яких виконувались встановлені обмеження на розміри площ функціональних зон, доводиться сканувати увесь простір можливих значень. Також, хоча PSO з модифікованим відбором працювало краще, ніж класичний PSO, однак його продуктивність за різними параметрами, окрім кількості досягнутих максимумів для схеми $5 \times 5$ була приблизно на рівні класичного GA. Крім того, модифікація мала найвищі затрати по часу роботи, бо так само як в класичному PSO, частинки в методі здійснювали пошук зі слабкою

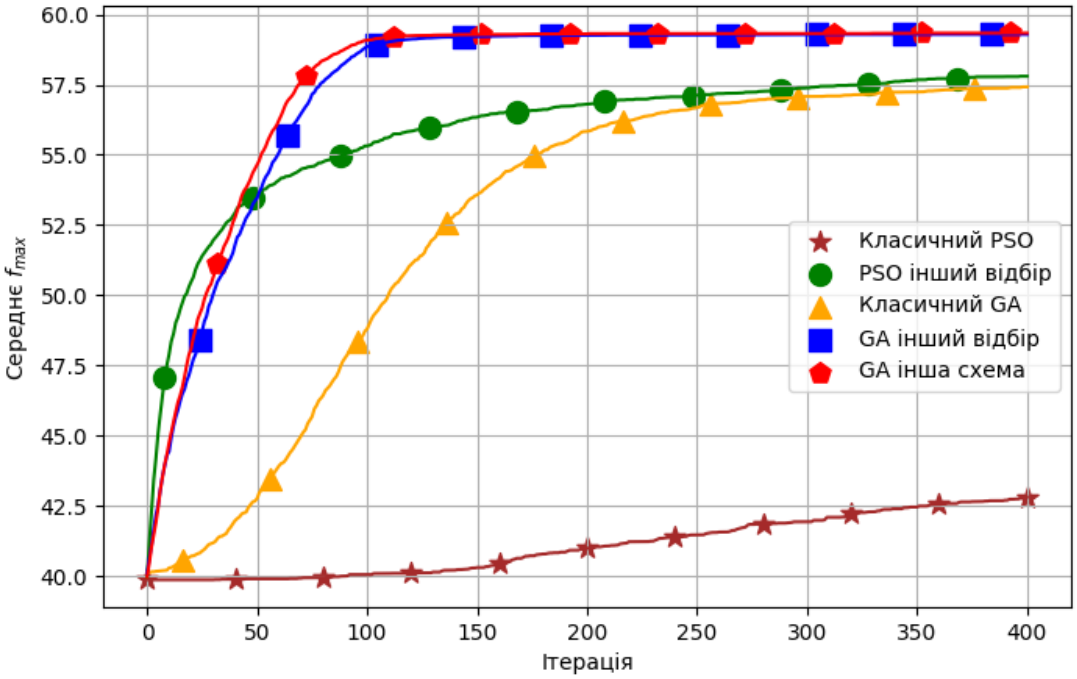



**Рисунок 3.1** – Графік досягнутих значень цільової функції по ітераціям для різних алгоритмів в середньому після 100 запусків для схеми розміром $3 \times 9$

ефективністю, часто зустрічаючи неприйнятні розв'язки, які доводилось постійно доброблювати, та які навіть після покращення залишались достатньо далекими від глобальних оптимумів.

На графіках з досягнутого найкращого значення по ітераціям (рис. 3.1, 3.2) видно, що PSO з модифікованим відбором має найкращі результати на ранніх етапах, однак пізніше метод застрягає в локальних оптимумах, що дуже сповільнює знаходження нових рішень. Водночас класичний GA показує протилежну поведінку: на ранніх ітераціях через суттєву різницю між хромосомами нові утворені рішення часто є неприйнятними, але з часом для всієї популяції вибудовується майбутній приблизний макет схеми, під час цього хромосоми стають більш схожими один на одного, і нові утворені розв'язки через схрещування приймаються частіше.

Аналізуючи швидкість збіжності з отриманим результатом

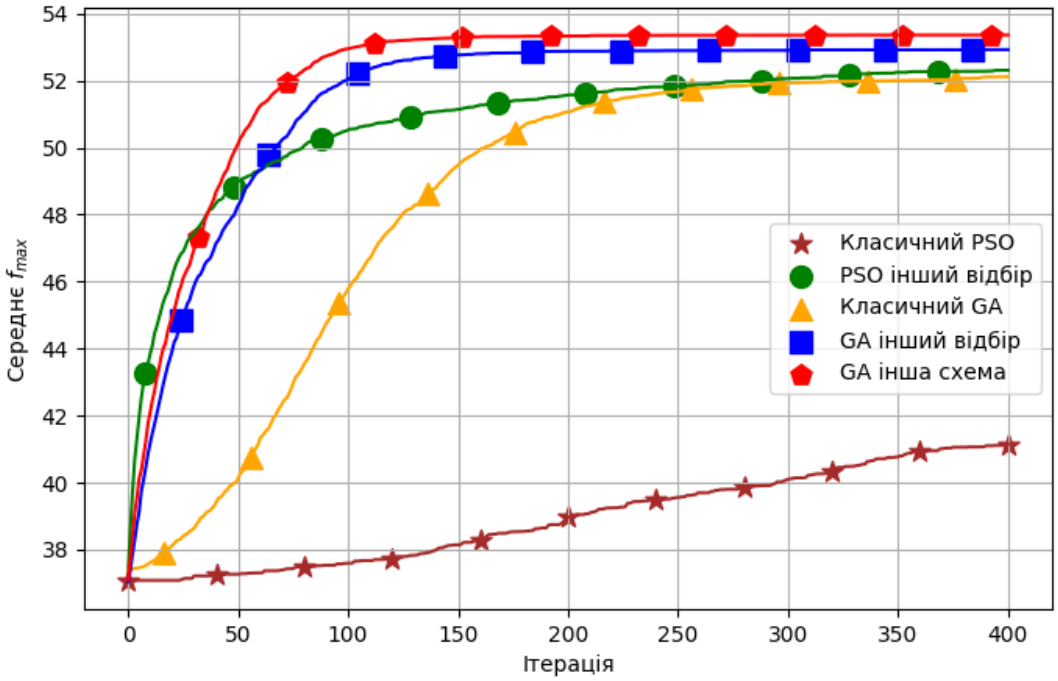



**Рисунок 3.2** – Графік досягнутих значень цільової функції по ітераціям для різних алгоритмів в середньому після 100 запусків для схеми розміром $5 \times 5$

(рис. 3.1, 3.2), то хоча PSO з модифікованим відбором та класичний GA показують деякий ріст протягом всіх ітерацій, однак після 300 ітерацій він вже можливий лише завдяки випадково знайденій вдалій модифікацій після багаторазового перебору, тому подальший ріст є несуттєвим та вже навряд чи досягне нових значень. Водночас GA з модифікованою схемою та GA з модифікованим відбором вже отримали значні результати ще до 200 ітерацій, тому для цих методів є допустимим зменшення максимальної кількості ітерацій до цього значення, і за рахунок цього можливе зменшення часу роботи цих алгоритмів приблизно в 2 рази без суттєвої втрати точності.

Оскільки значення кореляцій між зонами не змінювались для різних схем (рис. 3.3), то можна побачити, що в отриманих оптимальних рішеннях, крім зони транзиту та зони інформаційного обслуговування

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 0 | 0 | 0 | 1 | 1 | 4 | 5 |
| 3 | 3 | 0 | 0 | 0 | 1 | 1 | 4 | 5 |
| 3 | 3 | 0 | 0 | 0 | 0 | 2 | 2 | 6 |

**(а)**

| | | | | |
|---|---|---|---|---|
| 5 | 4 | 1 | 1 | 2 |
| 5 | 4 | 1 | 1 | 2 |
| 3 | 3 | 0 | 0 | 0 |
| 3 | 3 | 0 | 0 | 0 |
| 6 | 3 | 0 | 0 | 0 |

**(б)**

**Рисунок 3.3** – Схема оптимального розміщення функціональних зон логістичного парку аеропорт: (б) для схеми $3 \times 9$, (а) для схеми $5 \times 5$, де «0» — вантажна зона, «1» — митна зона, «2» — зона транзиту, «3» — зона з доданою вартістю, «4» — зона міжнародної торгівлі, «5» — допоміжна зона, «6» —зона інформаційного обслуговування

зберігається тенденція того, як та які зони межують між собою. Також на обох схемах видно, що деякі зони мають не зовсім прямокутний вигляд через можливе недоскональне визначення кількості клітинок на кожну зону. В такому випадку можна залишити отриману схему як є, якщо на виділеному місці можна вдало спланувати необхідні будівлі та технічні об'єкти, або можна за допомогою популяційних алгоритмів та методів переоптимізувати вже отримані розв'язки за рахунок зміни кількості клітинок для різних областей чи збільшенням розмірності сітки.

## Висновки до розділу 3

Було висвітлено, які основні параметри було обрано для математичної моделі авіаційного логістичного парку, та які розміри схем були прийняті для розгляду. Було висвітлено, як адаптовано генетичний оптимізатор та оптимізатор на основі рою частинок до поставленої задачі, які модифікації запропоновані для класичних методів. Усі методи було реалізовано та протестовано, основні результати чого були показані у

вигляді таблиці та рисунків та пізніше проаналізовано їх отримані значення цільової функції, характер збіжності та можливі особливості роботи методів, що спостерігались для даної задачі планування розміщення функціональних зон авіаційного логістичного парку.

Порівнюючи показані результати, то, не зважаючи на малу різницю в отриманих абсолютних значеннях цільової функції, дві з трьох запропонованих модифікацій, а саме GA з модифікованою схемою та GA з модифікованим відбором в рази частіше досягали глобальних оптимумів та отримували кращі результати як в середньому, так і при найгірших запусках, в порівнянні з класичним PSO та GA. Отримана третя модифікація, а саме вдосконалення PSO, хоча і показує покращений результат в порівнянні з класичним аналогом, однак не має жодної переваги в порівнянні з іншими модифікаціями та працює на рівні класичного GA, що ставить під сумнів доцільність такої модифікації.

# ВИСНОВКИ

Логістичні парки, до яких входять авіаційні логістичні парки, є складним та комплексним об'єктом, який має враховувати попит та пропозицію різних галузей, зважаючи на їх розташування, затрати на перевезення, зберігання та інші технічні параметри. Тому розробка ефективних планів парку, які б зменшували свої витрати, давали додатковий економічний ефект для підприємств, та разом з цим розробка та вдосконалення алгоритмів під задачу планування функціональних зон є складною і водночас важливою задачею у сфері логістичних поставок.

У даній роботі був проведений аналіз публікацій щодо загального стану логістичного парку аеропорту, було визначено основні функціональні області. За допомогою цього була розглянута математична модель на основі рівномірної сітки, що дозволяє ефективно розбити логістичний парк на окремі однакові блоки, на основі яких була запропонована цільова функція для розрахунку значень кореляцій різних функціональних зон для подальшої оптимізації всієї схеми планування авіаційного логістичного парку. Для розробки програмного забезпечення були розглянуті два популярні стохастичні методи з класу популяційних алгоритмів, а саме генетичний алгоритм та оптимізатор на основі рою частинок, та їх можливі варіації та модифікації. Також наприкінці було запропоновано та розроблено декілька модифікацій алгоритмів під умови задачі для декількох моделей логістичного парку аеропорту. Відповідні програмні реалізації було багаторазово протестовано для усереднення результату та проаналізовано з точки зору досягнутих результатів та часових затрат.

Отримані значення програмних реалізацій методів виявились кращими за класичний варіант генетичного алгоритма та методу на основі рою частинок, а саме: генетичний алгоритм з модифікованим відбором кращий на $1{,}3 - 3{,}9\%$ за середнім по усім запускам, кращий на

$2{,}8 - 8{,}0\%$ за найгіршим результатом, та в $3 - 6$ рази частіше знаходив глобальний максимум; генетичний алгоритм з модифікованою схемою кращий на $1{,}9 - 3{,}8\%$ за середнім по усім запускам, кращий на $3{,}2 - 8{,}3\%$ за найгіршим результатом, та в $5 - 22$ рази частіше знаходив максимум в залежності від обраної схеми. Часові затрати виявились більшими в 1,3 рази для генетичного алгоритму з модифікованим відбором та в 2,3 рази для генетичного алгоритму з модифікованою схемою, однак показані можливості щодо досягнення глобальних оптимумів перекривають цей недолік. Лише запропонований метод на основі рою частинок з модифікованим відбором показав обмежений результат з високими витратами по часу.

Враховуючи це, можна стверджувати, що за допомогою розробленого програмного забезпечення можна реалізовувати різні моделі схем розміщення авіаційного логістичного парку, та частина запропонованих алгоритмів є більш ефективними в порівнянні з класичними методами в побудові оптимальної схеми. Однак необхідно проводити подальші дослідження з вдосконалення часу роботи застосованих алгоритмів, тестування методів на більших схемах та схемах різних форм для аналізу їх продуктивності.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Tanweer Alam et al. “Genetic Algorithm: Reviews, Implementations, and Applications”. In: *arXiv preprint arXiv:2007.12673* (2020). DOI: 10.36227/techrxiv.12657173.

[2] Bushra Alhijawi and Arafat Awajan. “Genetic algorithms: Theory, genetic operators, solutions, and applications”. In: *Evolutionary Intelligence* 17.3 (2024), pp. 1245–1256.

[3] Beatriz Flamia Azevedo, Ana Maria AC Rocha, and Ana I Pereira. “Hybrid approaches to optimization and machine learning methods: a systematic literature review”. In: *Machine Learning* 113.7 (2024), pp. 4055–4097.

[4] Jianxin Chen. “Modeling and simulation analysis of optimal layout scheme of aviation logistics park based on genetic algorithm”. English. In: *Advanced Control for Applications: Engineering and Industrial Systems* 6 (2024). DOI: 10.1002/adc2.200.

[5] Joanna Alicja Dyczkowska and Olga Reshetnikova. “Logistics Centers in Ukraine: Analysis of the Logistics Center in Lviv”. In: *Energies* 15.21 (2022). ISSN: 1996-1073. DOI: 10.3390/en15217975. URL: https://www.mdpi.com/1996-1073/15/21/7975.

[6] Jiawen Fang, Canfei He, and Shengjun Zhu. “From death to birth: Do logistics parks help local renewals in logistics industry?” In: *Transport Policy* 158.C (2024), pp. 148–158. DOI: 10.1016/j.tranpol.2024.09. URL: https://ideas.repec.org/a/eee/trapol/v158y2024icp148-158.html.

[7] Ahmed G Gad. “Particle swarm optimization algorithm and its applications: a systematic review”. In: *Archives of computational methods in engineering* 29.5 (2022), pp. 2531–2561.

[8] Jingwen Gong. "Optimization of the layout of the air logistics park in M city". In: (2023).

[9] Hui Han. "A review and a model for logistics clusters". In: *International Journal of Logistics Systems and Management* 33 (2019), pp. 73 –96. DOI: 10.1504/IJLSM.2019.099662.

[10] Meetu Jain et al. "An Overview of Variants and Advancements of PSO Algorithm". In: *Applied Sciences* 12.17 (2022). ISSN: 2076-3417. DOI: 10.3390/app12178392. URL: https://www.mdpi.com/2076-3417/12/17/8392.

[11] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. "A review on genetic algorithm: past, present, and future". In: *Multimedia tools and applications* 80 (2021), pp. 8091–8126.

[12] Guoqi Li et al. "Planning versus the market: Logistics establishments and logistics parks in Chongqing, China". In: *Journal of Transport Geography* 82.C (2020). DOI: 10.1016/j.jtrangeo.2019.1. URL: https://ideas.repec.org/a/eee/jotrge/v82y2020ics096669231930448x.html.

[13] David A Liñán et al. "A hybrid deterministic-stochastic algorithm for the optimal design of process flowsheets with ordered discrete decisions". In: *Computers & Chemical Engineering* 180 (2024), p. 108501.

[14] Qingyu Luo et al. "A two-stage layout method for functional areas in logistics park". In: *Advances in Mechanical Engineering* 11.3 (2019), p. 1687814019829954. DOI: 10.1177/1687814019829954. eprint: https://doi.org/10.1177/1687814019829954. URL: https://doi.org/10.1177/1687814019829954.

[15] Geyu Ma. "Study on the layout planning of CDTF airport air logistics park based on SLP". In: (2023).

[16] Seyedali Mirjalili et al. "Genetic Algorithm: Theory, Literature Review, and Application in Image Reconstruction: Methods and Applications". In: 2020, pp. 69–85. ISBN: 978-3-030-12126-6. DOI: 10.1007/978-3-030-12127-3_5.

[17] Panos M Pardalos and H Edwin Romeijn. *Handbook of global optimization: Volume 2*. Vol. 62. Springer Science & Business Media, 2013.

[18] Tareq M Shami et al. "Particle swarm optimization: A comprehensive survey". In: *Ieee Access* 10 (2022), pp. 10031–10061.

[19] TO Ting et al. "Hybrid metaheuristic algorithms: past, present, and future". In: *Recent advances in swarm intelligence and evolutionary computation* (2015), pp. 71–83.

[20] Jingwen Wu. "Research on Layout Planning of Logistics Park of Zhengzhou Airport". In: (2023).

[21] Dan Yang et al. "Understanding the Effect of Multi-Agent Collaboration on the Performance of Logistics Park Projects: Evidence from China". In: *Sustainability* 14.7 (2022). ISSN: 2071-1050. DOI: 10 . 3390 / su14074179. URL: https://www.mdpi.com/2071-1050/14/7/4179.

[22] Dezhi Zhang, Richard Eglese, and Shuangyan Li. "Optimal location and size of logistics parks in a regional logistics network with economies of scale and CO2 emission taxes". In: *Transport* 33.1 (2018), pp. 52–68.

[23] Yijie Zhang. "Research on the layout optimization of aviation logistics park in Z city". In: (2024).

[24] Zhiqian Zheng. "Research on the Layout Planning of PD Airport Logistics Park Based on SLP". In: (2023).

[25] Nie Zilin, Duan Dongkun, and Yao Jiayi. "Study on the Planning of Airport Logistics Park in a City". In: *Proceedings of the 4th International Conference on Economics, Management, Law and Education (EMLE 2018)*. Atlantis Press, 2018, pp. 403–406. ISBN: 978-94-6252-639-6. DOI: 10 . 2991 / emle - 18 . 2018 . 73. URL: https://doi.org/10.2991/emle-18.2018.73.

# ДОДАТОК А ТЕКСТИ ПРОГРАМ

У додатку наведено програмний код методів та функцій, написаних на мові Python, що використовувся для знаходження моделі схеми розміщення авіаційного логістичного парку, для розрахунку цільової функції, для тестування та візуалізації отриманих результатів.

## А.1 Підключення бібліотек, визначення додаткових функцій

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from time import perf_counter
import pandas as pd

def function_for_array(scheme_array, corr_array, shape):
    n,m = shape
    fitness = np.zeros(scheme_array.shape[0])

    for i in range(0, n-1):
        fitness += corr_array[scheme_array[:,i,:],\
        scheme_array[:,i+1,:]].sum(axis=1)
    for j in range(0, m-1):
        fitness += corr_array[scheme_array[:,:,j],\
        scheme_array[:,:,j+1]].sum(axis=1)

    return fitness*2

def function_for_scheme(scheme_array, corr_array, shape):
    n,m = shape
    fitness = 0

    for i in range(0, n-1):
        fitness += corr_array[scheme_array[i,:],\
        scheme_array[i+1,:]].sum()
    for j in range(0, m-1):
        fitness += corr_array[scheme_array[:,j],\
        scheme_array[:,j+1]].sum()

    return fitness*2

colors = np.array([[0.2,0.2,  1],
                   [0.2,  1,0.2],
                   [0.4,0.4,0.4],
                   [  1,0.2,0.2],
                   [  0,0.7,0.7],
                   [0.7,0.7,  0],
```

```
                    [0.7,  0,0.7],])

def scheme_to_image(scheme, shape, colors):
    n,m = shape
    map_park = np.ones((n*6+1,m*6+1,3))
    i_tile = 1
    for i in range(n):
        j_tile = 1
        for j in range(m):
            map_park[i_tile:i_tile+5,j_tile:j_tile+5] = \
            np.ones((5,5,3))*colors[scheme[i,j]]
            j_tile += 6
        i_tile += 6
    return(map_park)
```

## A.2 Класичний GA

```
def gen_alg(shape, sizes_zones, count_pop, count_kids,
            prob_mutate, max_mutations, prob_crossover, function,
            corr_array, n_iter=100, seed=0):

    np.random.seed(seed)
    size = shape[0] * shape[1]

    initial_scheme = np.zeros(size, dtype=np.int8)
    code_zone = 0
    counter = 0
    for i in range(size):
        initial_scheme[i] = code_zone
        counter += 1
        if sizes_zones[code_zone] == counter:
            code_zone += 1
            counter = 0

    scheme_array = []
    for i in range(count_pop):
        scheme_array.append(np.random.permutation(initial_scheme))
    scheme_array = np.array(scheme_array, dtype=np.int8)

    fitness_array = function(scheme_array.reshape((-1,
    shape[0],shape[1])), corr_array, shape)

    best_iter_scheme = []
    best_iter_fitness = []

    best_scheme = fitness_array.argmax()
    best_iter_scheme.append(scheme_array[best_scheme])
    best_iter_fitness.append(fitness_array[best_scheme])

    for iter in range(n_iter):

        prob = fitness_array/fitness_array.sum()
        parents_array = scheme_array[np.random.choice(count_pop,
        size=count_kids, replace=True, p=prob)]

        kids_array = parents_array.copy()
```

```
        for i in range(0,count_kids-1,2):

            is_cross = np.random.choice([0,1], size=1,
            p=[1-prob_crossover, prob_crossover])
            if is_cross:

                a1, a2 = np.random.randint(size, size=2)
                if a1 > a2:
                    a1, a2 = a2, a1

                mask_crossover = np.zeros(size, dtype=np.bool)
                mask_crossover[a1:a2+1] = 1
                kids_array[i  ][mask_crossover] =\
                parents_array[i+1][mask_crossover]
                kids_array[i+1][mask_crossover] =\
                parents_array[i  ][mask_crossover]

                is_mutated = np.random.choice([0,1], size=2,
                p=[1-prob_mutate, prob_mutate])

            for j in range(2):
                if (is_mutated[j]):
                    n_mutation = np.random.randint(2,
                    max_mutations+1, size=1)
                    mask = np.random.choice(size,
                    size=n_mutation, replace=False)
                    kids_array[i+j][mask] =\
                    np.random.permutation(kids_array[i+j][mask])

        kids_fitness_array = function(kids_array.reshape((-1,
        shape[0],shape[1])), corr_array, shape)

        for i in range(count_kids):
            unique, counts = np.unique(kids_array[i], return_counts=True)
            penalty = sizes_zones.copy()
            penalty[unique] -= counts
            kids_fitness_array[i] -= n_zones*(penalty**2).sum()

        scheme_array = np.append(scheme_array, kids_array, axis=0)
        fitness_array = np.append(fitness_array, kids_fitness_array,
        axis=0)

        index_sort = fitness_array.argsort()[-count_pop:]
        scheme_array = scheme_array[index_sort]
        fitness_array = fitness_array[index_sort]

        best_scheme = fitness_array.argmax()
        best_iter_scheme.append(scheme_array[best_scheme])
        best_iter_fitness.append(fitness_array[best_scheme])

    return np.array(best_iter_scheme), np.array(best_iter_fitness)
```

## A.3 Класичний PSO

```
def pso_alg(shape, sizes_zones, x_min, x_max, c1, c2, v_max, v_min,
```

```
        count_pop, function, corr_array, n_iter=100, seed=0):

size = shape[0] * shape[1]
n_zones = sizes_zones.shape[0]
np.random.seed(seed)

x = np.random.random((count_pop,size))*(n_zones-1)

round_x = np.round(x).astype(int)

fitness_array = function(round_x.reshape((-1,
shape[0],shape[1])), corr_array, shape)

for i in range(count_pop):
    unique, counts = np.unique(round_x[i], return_counts=True)
    penalty = sizes_zones.copy()
    penalty[unique] -= counts
    fitness_array[i] -= 3*(penalty**2).sum()

p_best = round_x.copy()
p_fitness = fitness_array.copy()
g_best = p_best[p_fitness.argmax()]
v = np.zeros((count_pop,size))

best_iter_scheme = []
best_iter_fitness = []

best_scheme = fitness_array.argmax()
best_iter_scheme.append(x[best_scheme])
best_iter_fitness.append(fitness_array[best_scheme])

for iter in range(n_iter):

    r1 = np.random.random((count_pop,size))
    r2 = np.random.random((count_pop,size))
    v += c1*r1*(p_best - x) + c2*r2*(g_best - x)
    v = limit_value(v, v_min, v_max)
    x += v
    x = limit_value(x, x_min, x_max)

    round_x = np.round(x).astype(int)

    fitness_array = function(round_x.reshape((-1,
    shape[0],shape[1])), corr_array, shape)

    for i in range(count_pop):
        unique, counts = np.unique(round_x[i], return_counts=True)
        penalty = sizes_zones.copy()
        penalty[unique] -= counts
        fitness_array[i] -= 3*(penalty**2).sum()

    improved_p = p_fitness < fitness_array

    if improved_p.sum() > 0:
        p_best[improved_p] = round_x[improved_p]
        p_fitness[improved_p] = fitness_array[improved_p]
        g_best = p_best[p_fitness.argmax()]

    best_iter_scheme.append(g_best)
    best_iter_fitness.append(p_fitness.max())
```

```
    return np.array(best_iter_scheme), np.array(best_iter_fitness)
```

## A.4 Модифікація відбору для GA

```
def improve_gen_alg(shape, sizes_zones, count_pop, count_kids,
            prob_mutate, max_mutations, prob_crossover, function,
            corr_array, n_iter=100, seed=0):

    np.random.seed(seed)
    n_zones = sizes_zones.shape[0]
    size = shape[0] * shape[1]

    initial_scheme = np.zeros(size, dtype=np.int8)
    code_zone = 0
    counter = 0
    for i in range(size):
        initial_scheme[i] = code_zone
        counter += 1
        if sizes_zones[code_zone] == counter:
            code_zone += 1
            counter = 0

    scheme_array = []
    for i in range(count_pop):
        scheme_array.append(np.random.permutation(initial_scheme))
    scheme_array = np.array(scheme_array, dtype=np.int8)

    fitness_array = function(scheme_array.reshape((-1,
    shape[0],shape[1])), corr_array, shape)

    best_iter_scheme = []
    best_iter_fitness = []

    best_scheme = fitness_array.argmax()
    best_iter_scheme.append(scheme_array[best_scheme])
    best_iter_fitness.append(fitness_array[best_scheme])

    for iter in range(n_iter):

        prob = fitness_array/fitness_array.sum()
        parents_array = scheme_array[np.random.choice(count_pop,
        size=count_kids, replace=True, p=prob)]

        kids_array = parents_array.copy()

        for i in range(0,count_kids-1,2):

            is_cross = np.random.choice([0,1], size=1,
            p=[1-prob_crossover, prob_crossover])
            if is_cross:

                a1, a2 = np.random.randint(size, size=2)
                if a1 > a2:
                    a1, a2 = a2, a1
```

```
            mask_crossover = np.zeros(size, dtype=np.bool)
            mask_crossover[a1:a2+1] = 1
            kids_array[i  ][mask_crossover] =\
            parents_array[i+1][mask_crossover]
            kids_array[i+1][mask_crossover] =\
            parents_array[i  ][mask_crossover]

        is_mutated = np.random.choice([0,1], size=2,
        p=[1-prob_mutate, prob_mutate])

        for j in range(2):
            if (is_mutated[j]):
                n_mutation = np.random.randint(2,
                max_mutations+1, size=1)
                mask = np.random.choice(size,
                size=n_mutation, replace=False)
                kids_array[i+j][mask] =\
                np.random.permutation(kids_array[i+j][mask])

    improve_kids = kids_array.copy()
    for c in range(count_kids):
        unique, counts = np.unique(improve_kids[c],
        return_counts=True)
        penalty = sizes_zones.copy()
        penalty[unique] -= counts
        if (penalty**2).sum()>0:
            mask = np.zeros(size, dtype=bool)
            for j in range(n_zones):
                if penalty[j] < 0:
                    mask[np.random.choice(np.arange(size)[
                    improve_kids[c] == j], size=(-penalty[j]),
                    replace=False)] = 1
            need_zones = np.arange(n_zones)[penalty > 0]
            cn_zones = penalty[penalty > 0]
            new_scheme = []
            code_zone = 0
            counter = 0
            for i in range(cn_zones.sum()):
                new_scheme.append(need_zones[code_zone])
                counter += 1
                if cn_zones[code_zone] == counter:
                    code_zone += 1
                    counter = 0
            improve_kids[c][mask] = np.random.permutation(new_scheme)

    kids_fitness_array =\
    function(improve_kids.reshape((-1,shape[0],shape[1])),
    corr_array, shape)

    scheme_array = np.append(scheme_array, improve_kids, axis=0)
    fitness_array = np.append(fitness_array, kids_fitness_array,
    axis=0)

    index_sort = fitness_array.argsort()[-count_pop:]
    scheme_array = scheme_array[index_sort]
    fitness_array = fitness_array[index_sort]

    best_scheme = fitness_array.argmax()
    best_iter_scheme.append(scheme_array[best_scheme])
    best_iter_fitness.append(fitness_array[best_scheme])
```

```
    return np.array(best_iter_scheme), np.array(best_iter_fitness)
```

## A.5 Модифікація відбору для PSO

```
def improve_pso_alg(shape, sizes_zones, x_min, x_max, c1, c2,
    v_max, v_min, count_pop, function, corr_array, n_iter=100, seed=0):

    size = shape[0] * shape[1]
    n_zones = sizes_zones.shape[0]
    np.random.seed(seed)

    def limit_value(x, x_min, x_max):
        max_mask = x > x_max
        x[max_mask] = (max_mask*x_max)[max_mask]
        min_mask = x < x_min
        x[min_mask] = (min_mask*x_min)[min_mask]
        return x

    initial_scheme = np.zeros(size, dtype=np.int8)
    code_zone = 0
    counter = 0
    for i in range(size):
        initial_scheme[i] = code_zone
        counter += 1
        if sizes_zones[code_zone] == counter:
            code_zone += 1
            counter = 0

    x = []
    for i in range(count_pop):
        x.append(np.random.permutation(initial_scheme))
    x = np.array(x, dtype=float)

    fitness_array = function(np.round(x).astype(int)\
    .reshape((-1,shape[0],shape[1])), corr_array, shape)

    p_best = np.round(x).astype(int)
    p_fitness = fitness_array.copy()
    g_best = p_best[p_fitness.argmax()]
    v = np.zeros((count_pop,size))

    best_iter_scheme = []
    best_iter_fitness = []

    best_scheme = fitness_array.argmax()
    best_iter_scheme.append(x[best_scheme])
    best_iter_fitness.append(fitness_array[best_scheme])

    for iter in range(n_iter):

        r1 = np.random.random((count_pop,size))
        r2 = np.random.random((count_pop,size))
        v += c1*r1*(p_best - x) + c2*r2*(g_best - x)
        v = limit_value(v, v_min, v_max)
        x += v
```

```
        x = limit_value(x, x_min, x_max)

        improve_x = np.round(x).astype(int)
        for c in range(count_pop):
            unique, counts = np.unique(improve_x[c], return_counts=True)
            penalty = sizes_zones.copy()
            penalty[unique] -= counts
            if (penalty**2).sum()>0:
                mask = np.zeros(size, dtype=bool)
                for j in range(n_zones):
                    if penalty[j] < 0:
                        mask[np.random.choice(np.arange(size)[
                        improve_x[c] == j], size=(-penalty[j]),
                        replace=False)] = 1
                need_zones = np.arange(n_zones)[penalty > 0]
                cn_zones = penalty[penalty > 0]
                new_scheme = []
                code_zone = 0
                counter = 0
                for i in range(cn_zones.sum()):
                    new_scheme.append(need_zones[code_zone])
                    counter += 1
                    if cn_zones[code_zone] == counter:
                        code_zone += 1
                        counter = 0
                improve_x[c][mask] = np.random.permutation(new_scheme)

        fitness_array =\
        function(improve_x.reshape((-1,shape[0],shape[1])),
        corr_array, shape)

        improved_p = p_fitness < fitness_array

        if improved_p.sum() > 0:
            p_best[improved_p] = improve_x[improved_p]
            p_fitness[improved_p] = fitness_array[improved_p]
            g_best = p_best[p_fitness.argmax()]

        best_iter_scheme.append(g_best)
        best_iter_fitness.append(p_fitness.max())

    return np.array(best_iter_scheme), np.array(best_iter_fitness)
```

## A.6 Модифікація схеми для GA

```
def gen_alg_other_scheme(shape, sizes_zones, count_pop,
    count_kids, prob_mutate, max_mutations, prob_crossover, function,
    corr_array, n_iter=100, seed=0):

    size = shape[0] * shape[1]
    n_zones = sizes_zones.shape[0]
    np.random.seed(seed)

    first_scheme = np.arange(size)
    scheme_array = []
    for i in range(count_pop):
```

```
        scheme_array.append(np.random.permutation(first_scheme))
    scheme_array = np.array(scheme_array, dtype=np.int8)

    def get_initial_scheme(scheme, sizes_zones):
        size = scheme.shape[0]
        initial_scheme = np.zeros(size, dtype=np.int8)
        code_zone = 0
        counter = 0
        for i in range(size):
            initial_scheme[scheme[i]] = code_zone
            counter += 1
            if sizes_zones[code_zone] == counter:
                code_zone += 1
                counter = 0
        return initial_scheme

    initial_array = np.zeros((count_pop,size), dtype=np.int16)
    initial_array = np.zeros((count_kids,size), dtype=np.int16)
    for i in range(count_pop):
        initial_array[i] = get_initial_scheme(scheme_array[i],
        sizes_zones)
    fitness_array = function(initial_array.reshape((-1,shape[0],
    shape[1])), corr_array, shape)

    best_iter_scheme = []
    best_iter_fitness = []

    best_scheme = fitness_array.argmax()
    best_iter_scheme.append(scheme_array[best_scheme])
    best_iter_fitness.append(fitness_array[best_scheme])

    for iter in range(n_iter):

        prob = fitness_array/fitness_array.sum()
        parents_array = scheme_array[np.random.choice(count_pop,
        size=count_kids, replace=True, p=prob)]

        kids_array = parents_array.copy()

        for i in range(0,count_kids-1,2):

            is_cross = np.random.choice([0,1], size=1,
            p=[1-prob_crossover, prob_crossover])
            if is_cross:

                a1, a2 = np.random.randint(size, size=2)
                if a1 > a2:
                    a1, a2 = a2, a1

                mask_crossover = np.zeros(size, dtype=np.bool)
                mask_crossover[a1:a2+1] = 1
                outside_mask = (1-mask_crossover).astype(bool)
                inner_parts = np.array([parents_array[i
][mask_crossover],
                parents_array[i+1][mask_crossover]])

                for j in range(2):

                    outside = parents_array[i+1-j][outside_mask]
                    extra =   parents_array[i+j  ][outside_mask]
```

```
                    for e in range(outside.shape[0]):
                        check_element = outside[e]
                        if (check_element in inner_parts[j]):
                            outside[e] = -1
                        else:
                            extra = extra[extra!=check_element]
                    outside[outside==-1] = np.random.permutation(extra)
                    kids_array[i+j][outside_mask] = outside

            is_mutated = np.random.choice([0,1], size=2, p=[1-prob_mutate,
            prob_mutate])

            for j in range(2):
                if (is_mutated[j]):
                    n_mutation = np.random.randint(2, max_mutations+1,
                    size=1)
                    mask = np.random.choice(size,
                    size=n_mutation, replace=False)
                    kids_array[i+j][mask] = np.random.permutation(
                    kids_array[i+j][mask])

        for i in range(count_kids):
            initial_array[i] = get_initial_scheme(kids_array[i],
            sizes_zones)
        kids_fitness_array = function(initial_array.reshape((-1,
        shape[0],shape[1])), corr_array, shape)

        scheme_array = np.append(scheme_array, kids_array, axis=0)
        fitness_array = np.append(fitness_array, kids_fitness_array,
        axis=0)

        index_sort = fitness_array.argsort()[-count_pop:]
        scheme_array = scheme_array[index_sort]
        fitness_array = fitness_array[index_sort]

        best_scheme = fitness_array.argmax()
        best_iter_scheme.append(get_initial_scheme(
        scheme_array[best_scheme],sizes_zones))
        best_iter_fitness.append(fitness_array[best_scheme])

    return np.array(best_iter_scheme), np.array(best_iter_fitness)
```

## A.7 Комбінація PSO та GA з модифікованим відбором

```
def gen_pso_alg(shape, sizes_zones, x_min, x_max, c1, c2,
    v_max, v_min, count_pop, count_kids, prob_mutate, max_mutations,
    prob_crossover, function, corr_array, n_iter=100, seed=0):

    size = shape[0] * shape[1]
    n_zones = sizes_zones.shape[0]
    half_pop = count_pop//2
    np.random.seed(seed)

    borders_zones = np.zeros(n_zones+1, dtype=np.int8)
    borders_zones[-1] = size
```

```
for i in range(1, n_zones):
    borders_zones[i] = borders_zones[i-1] + sizes_zones[i-1]

def limit_value(x, x_min, x_max):
    max_mask = x > x_max
    x[max_mask] = (max_mask*x_max)[max_mask]
    min_mask = x < x_min
    x[min_mask] = (min_mask*x_min)[min_mask]
    return x

initial_scheme = np.zeros(size, dtype=np.int8)
code_zone = 0
counter = 0
for i in range(size):
    initial_scheme[i] = code_zone
    counter += 1
    if sizes_zones[code_zone] == counter:
        code_zone += 1
        counter = 0
x = []
for i in range(count_pop):
    x.append(np.random.permutation(initial_scheme))
x = np.array(x, dtype=float)

fitness_array = function(x.round().astype(int).reshape((-1,
shape[0],shape[1])), corr_array, shape)

p_best = x.copy().round().astype(int)
p_fitness = fitness_array.copy()
g_best = p_best[p_fitness.argmax()]
v = np.zeros((count_pop,size))

best_iter_scheme = []
best_iter_fitness = []

best_scheme = fitness_array.argmax()
best_iter_scheme.append(x[best_scheme])
best_iter_fitness.append(fitness_array[best_scheme])

def upgrade_p(scheme_array, fitness_array, shape, n_zones,
    sizes_zones, size, prob_mutate, max_mutations, function,
    corr_array, calculate_fitness, borders_zones, iter, count_kids):

    size_array = scheme_array.shape[0]

    prob = fitness_array/fitness_array.sum()
    parents_array = scheme_array[np.random.choice(size_array,
    size=count_kids, replace=True, p=prob)]

    kids_array = parents_array.copy()

    for i in range(0,count_kids-1,2):

        is_cross = np.random.choice([0,1], size=1,
        p=[1-prob_crossover, prob_crossover])
        if is_cross:

            a1, a2 = np.random.randint(size, size=2)
            if a1 > a2:
                a1, a2 = a2, a1
```

```
                mask_crossover = np.zeros(size, dtype=np.bool)
                mask_crossover[a1:a2+1] = True
                kids_array[i  ][mask_crossover] =\
                parents_array[i+1][mask_crossover]
                kids_array[i+1][mask_crossover] =\
                parents_array[i  ][mask_crossover]

            is_mutated = np.random.choice([0,1], size=2, p=[1-prob_mutate,
            prob_mutate])

            for j in range(2):
                if is_mutated[j]:
                    n_mutation = np.random.randint(2, max_mutations+1,
                    size=1)
                    mask = np.random.choice(size,
                    size=n_mutation, replace=False)
                    kids_array[i+j][mask] = np.random.permutation(
                    kids_array[i+j][mask])

        improve_kids = kids_array.copy()
        for c in range(count_kids):
            unique, counts = np.unique(improve_kids[c],
            return_counts=True)
            penalty = sizes_zones.copy()
            penalty[unique] -= counts
            if (penalty**2).sum()>0:
                mask = np.zeros(size, dtype=bool)
                for j in range(n_zones):
                    if penalty[j] < 0:
                        mask[np.random.choice(np.arange(size)[
improve_kids[c]==j],size=(-penalty[j]),replace=False)]=1
                need_zones = np.arange(n_zones)[penalty > 0]
                cn_zones = penalty[penalty > 0]
                new_scheme = []
                code_zone = 0
                counter = 0
                for i in range(cn_zones.sum()):
                    new_scheme.append(need_zones[code_zone])
                    counter += 1
                    if cn_zones[code_zone] == counter:
                        code_zone += 1
                        counter = 0
                improve_kids[c][mask] = np.random.permutation(new_scheme)

        kids_fitness_array = function(improve_kids.reshape((-1,
        shape[0],shape[1])), corr_array, shape)

        scheme_array = np.append(scheme_array, improve_kids, axis=0)
        fitness_array = np.append(fitness_array, kids_fitness_array,
        axis=0)

        index_sort = fitness_array.argsort()[-size_array:]
        scheme_array = scheme_array[index_sort]
        fitness_array = fitness_array[index_sort]

        return scheme_array, fitness_array

    for iter in range(n_iter):
```

```
        r1 = np.random.random((count_pop,size))
        r2 = np.random.random((count_pop,size))
        v += c1*r1*(p_best - x) + c2*r2*(g_best - x)
        v = limit_value(v, v_min, v_max)
        x += v
        x = limit_value(x, x_min, x_max)

        improve_x = x.copy().round().astype(int)
        for c in range(count_pop):
            unique, counts = np.unique(improve_x[c], return_counts=True)
            penalty = sizes_zones.copy()
            penalty[unique] -= counts
            if (penalty**2).sum()>0:
                mask = np.zeros(size, dtype=bool)
                for j in range(n_zones):
                    if penalty[j] < 0:
                        mask[np.random.choice(np.arange(size)[
improve_x[c]==j],size=(-penalty[j]),replace=False)]=1
                need_zones = np.arange(n_zones)[penalty > 0]
                cn_zones = penalty[penalty > 0]
                initial_scheme = []
                code_zone = 0
                counter = 0
                for i in range(cn_zones.sum()):
                    initial_scheme.append(need_zones[code_zone])
                    counter += 1
                    if cn_zones[code_zone] == counter:
                        code_zone += 1
                        counter = 0
                improve_x[c][mask] = np.random.permutation(initial_scheme)

        fitness_array = function(improve_x.reshape((-1,
        shape[0],shape[1])), corr_array, shape)

        improved_p = p_fitness < fitness_array

        if improved_p.sum() > 0:
            p_best[improved_p] = improve_x[improved_p]
            p_fitness[improved_p] = fitness_array[improved_p]
            g_best = p_best[p_fitness.argmax()]

        index_sort = p_fitness.argsort()[:half_pop]
        p_best[index_sort], p_fitness[index_sort] =\
        upgrade_p(p_best[index_sort], p_fitness[index_sort], shape,
        n_zones, sizes_zones, size, prob_mutate, max_mutations,
        function, corr_array, function, borders_zones, iter, count_kids)
        g_best = p_best[p_fitness.argmax()]

        best_iter_scheme.append(g_best)
        best_iter_fitness.append(p_fitness.max())

    return np.array(best_iter_scheme), np.array(best_iter_fitness)
```

## A.8 Комбінація PSO з модифікованим відбором та GA з модифікованою схемою

```
def gen_pso_alg_other_scheme(shape, sizes_zones, x_min, x_max, c1,
    c2, v_max, v_min,count_pop, count_kids, prob_mutate, max_mutations,
    prob_crossover, function, corr_array, n_iter=100, seed=0):

    size = shape[0] * shape[1]
    n_zones = sizes_zones.shape[0]
    half_pop = count_pop//2
    np.random.seed(seed)

    borders_zones = np.zeros(n_zones+1, dtype=np.int8)
    borders_zones[-1] = size
    for i in range(1, n_zones):
        borders_zones[i] = borders_zones[i-1] + sizes_zones[i-1]

    def limit_value(x, x_min, x_max):
        max_mask = x > x_max
        x[max_mask] = (max_mask*x_max)[max_mask]
        min_mask = x < x_min
        x[min_mask] = (min_mask*x_min)[min_mask]
        return x

    initial_scheme = np.zeros(size, dtype=np.int8)
    code_zone = 0
    counter = 0
    for i in range(size):
        initial_scheme[i] = code_zone
        counter += 1
        if sizes_zones[code_zone] == counter:
            code_zone += 1
            counter = 0
    x = []
    for i in range(count_pop):
        x.append(np.random.permutation(initial_scheme))
    x = np.array(x, dtype=float)

    fitness_array = function(x.round().astype(int).reshape((-1,
    shape[0],shape[1])), corr_array, shape)

    p_best = x.copy().round().astype(int)
    p_fitness = fitness_array.copy()
    g_best = p_best[p_fitness.argmax()]
    v = np.zeros((count_pop,size))

    best_iter_scheme = []
    best_iter_fitness = []

    best_scheme = fitness_array.argmax()
    best_iter_scheme.append(x[best_scheme])
    best_iter_fitness.append(fitness_array[best_scheme])

    def get_initial_scheme(scheme, sizes_zones):
        size = scheme.shape[0]
        n_zones = sizes_zones.shape[0]
        initial_scheme = np.zeros(size, dtype=np.int8)
        code_zone = 0
        counter = 0
        for i in range(size):
            initial_scheme[scheme[i]] = code_zone
            counter += 1
            if sizes_zones[code_zone] == counter:
```

```
                code_zone += 1
                counter = 0
        return initial_scheme

    def get_reverse_scheme(scheme, sizes_zones, borders_zones):
        size = scheme.shape[0]
        reverse_scheme = np.zeros(size, dtype=np.int8)
        for i in range(size):
            reverse_scheme[borders_zones[scheme[i]]] = i
            borders_zones[scheme[i]] += 1
        return reverse_scheme

    def upgrade_p(scheme_array, fitness_array, shape, n_zones,
        sizes_zones, size, prob_mutate, max_mutations, function,
        corr_array, calculate_fitness, borders_zones, iter, count_kids):

        size_array = scheme_array.shape[0]

        prob = fitness_array/fitness_array.sum()
        parents_array = scheme_array[np.random.choice(size_array,
        size=count_kids, replace=True, p=prob)]

        for i in range(count_kids):
            parents_array[i] = get_reverse_scheme(parents_array[i],
            sizes_zones, borders_zones.copy())

        kids_array = parents_array.copy()

        for i in range(0,count_kids-1,2):

            is_cross = np.random.choice([0,1], size=1,
            p=[1-prob_crossover, prob_crossover])
            if is_cross:

                a1, a2 = np.random.randint(size, size=2)
                if a1 > a2:
                    a1, a2 = a2, a1

                mask_crossover = np.zeros(size, dtype=np.bool)
                mask_crossover[a1:a2+1] = 1

                outside_mask = (1-mask_crossover).astype(bool)
                inner_parts = np.array([parents_array[i
][mask_crossover],
                parents_array[i+1][mask_crossover]])

                for j in range(2):

                    outside = parents_array[i+1-j][outside_mask]
                    extra =   parents_array[i+j  ][outside_mask]

                    for e in range(outside.shape[0]):
                        check_element = outside[e]
                        if (check_element in inner_parts[j]):
                            outside[e] = -1
                        else:
                            extra = extra[extra!=check_element]
                    outside[outside==-1] = np.random.permutation(extra)
                    kids_array[i+j][outside_mask] = outside
```

```
                is_mutated = np.random.choice([0,1], size=2,
                p=[1-prob_mutate, prob_mutate])

                for j in range(2):
                    if is_mutated[j]:
                        n_mutation = np.random.randint(2, max_mutations+1,
                        size=1)
                        mask = np.random.choice(size,
                        size=n_mutation, replace=False)
                        kids_array[i+j][mask] = np.random.permutation(
                        kids_array[i+j][mask])

        for i in range(count_kids):
            kids_array[i] = get_initial_scheme(kids_array[i], sizes_zones)

        kids_fitness_array = calculate_fitness(kids_array.reshape((-1,
        shape[0],shape[1])), corr_array, shape)

        scheme_array = np.append(scheme_array, kids_array, axis=0)
        fitness_array = np.append(fitness_array, kids_fitness_array,
        axis=0)

        index_sort = fitness_array.argsort()[-size_array:]
        scheme_array = scheme_array[index_sort]
        fitness_array = fitness_array[index_sort]

        return scheme_array, fitness_array

    for iter in range(n_iter):

        r1 = np.random.random((count_pop,size))
        r2 = np.random.random((count_pop,size))
        v += c1*r1*(p_best - x) + c2*r2*(g_best - x)
        v = limit_value(v, v_min, v_max)
        x += v
        x = limit_value(x, x_min, x_max)

        improve_x = x.copy().round().astype(int)
        for c in range(count_pop):
            unique, counts = np.unique(improve_x[c], return_counts=True)
            penalty = sizes_zones.copy()
            penalty[unique] -= counts
            if (penalty**2).sum()>0:
                mask = np.zeros(size, dtype=bool)
                for j in range(n_zones):
                    if penalty[j] < 0:
                        mask[np.random.choice(np.arange(size)[
improve_x[c]==j],size=(-penalty[j]),replace=False)]=1
                need_zones = np.arange(n_zones)[penalty > 0]
                cn_zones = penalty[penalty > 0]
                initial_scheme = []
                code_zone = 0
                counter = 0
                for i in range(cn_zones.sum()):
                    initial_scheme.append(need_zones[code_zone])
                    counter += 1
                    if cn_zones[code_zone] == counter:
                        code_zone += 1
                        counter = 0
                improve_x[c][mask] = np.random.permutation(initial_scheme)
```

```
        fitness_array = function(improve_x.reshape((-1,
        shape[0],shape[1])), corr_array, shape)

        improved_p = p_fitness < fitness_array

        if improved_p.sum() > 0:
            p_best[improved_p] = improve_x[improved_p]
            p_fitness[improved_p] = fitness_array[improved_p]
            g_best = p_best[p_fitness.argmax()]

        index_sort = p_fitness.argsort()[:half_pop]
        p_best[index_sort], p_fitness[index_sort] = upgrade_p(
        p_best[index_sort], p_fitness[index_sort], shape,  n_zones,
        sizes_zones, size, prob_mutate, max_mutations, function,
        corr_array, function, borders_zones, iter, count_kids)
        g_best = p_best[p_fitness.argmax()]

        best_iter_scheme.append(g_best)
        best_iter_fitness.append(p_fitness.max())

    return np.array(best_iter_scheme), np.array(best_iter_fitness)
```

## A.9 Тестування методів

```
corr_array = np.array([[   1, 0.4, 0.3, 0.2, 0.2,   0,   0],
                       [ 0.4,   1, 0.3, 0.1, 0.2,   0,   0],
                       [ 0.3, 0.3,   1, 0.1, 0.1,   0, 0.1],
                       [ 0.2, 0.1, 0.1,   1,   0,   0, 0.1],
                       [ 0.2, 0.2, 0.1,   0,   1, 0.1,   0],
                       [   0,   0,   0,   0, 0.1,   1,-0.1],
                       [   0,   0, 0.1, 0.1,   0,-0.1,   1]])
sizes_zones = np.array([10,4,2,6,2,2,1])
shape = np.array([3,9])
size = shape[0] * shape[1]
count_pop = 600
count_kids = 300
prob_mutate = 0.3
prob_crossover = 1.0
max_mutations = 5
n_iter = 400
x_min = np.ones(size)*(0)
x_max = np.ones(size)*(6)
c1 = 2
c2 = 1
v_max = np.ones(size)*(1.0)
v_min = np.ones(size)*(-1.0)

repeats = 100
results = np.zeros((7,repeats,n_iter+1))
np.random.seed(1)
seeds = np.random.choice(10000000, size=7*repeats, replace=False)
time_work = np.zeros(7)


for i in range(repeats):
```

```
        print(f"{round(i/repeats*100)} %")

        start_time = perf_counter()
        scheme, results[0,i] = pso_alg(shape, sizes_zones, x_min, x_max,
        c1, c2, v_max, v_min, count_pop, function_for_array, corr_array,
        n_iter, seed=seeds[i])
        time_work[0] += (perf_counter()-start_time)

        start_time = perf_counter()
        scheme, results[1,i] = improve_pso_alg(shape, sizes_zones,
        x_min, x_max, c1, c2, v_max, v_min, count_pop, function_for_array,
        corr_array, n_iter, seed=seeds[i+repeats])
        time_work[1] += (perf_counter()-start_time)

        start_time = perf_counter()
        scheme, results[2,i] = gen_alg(shape, sizes_zones, count_pop,
        count_kids, prob_mutate, max_mutations, 0.9, function_for_array,
        corr_array, n_iter, seed=seeds[i+repeats*2])
        time_work[2] += (perf_counter()-start_time)

        start_time = perf_counter()
        scheme, results[3,i] = improve_gen_alg(shape, sizes_zones, count_pop,
        count_kids, prob_mutate, max_mutations, prob_crossover,
        function_for_array, corr_array, n_iter, seed=seeds[i+repeats*3])
        time_work[3] += (perf_counter()-start_time)

        start_time = perf_counter()
        scheme, results[4,i] = gen_alg_other_scheme(shape, sizes_zones,
        count_pop, count_kids, prob_mutate, max_mutations, prob_crossover,
        function_for_array, corr_array, n_iter, seed=seeds[i+repeats*4])
        time_work[4] += (perf_counter()-start_time)

        start_time = perf_counter()
        scheme, results[5,i] = gen_pso_alg(shape, sizes_zones,
        x_min, x_max, c1, c2, v_max, v_min, count_pop, count_kids,
        prob_mutate, max_mutations, prob_crossover, function_for_array,
        corr_array, n_iter, seed=seeds[i+repeats*5])
        time_work[5] += (perf_counter()-start_time)

        start_time = perf_counter()
        scheme, results[6,i] = gen_pso_alg_other_scheme(shape, sizes_zones,
        x_min, x_max, c1, c2, v_max, v_min, count_pop, count_kids,
        prob_mutate, max_mutations, prob_crossover, function_for_array,
        corr_array, n_iter, seed=seeds[i+repeats*6])
        time_work[6] += (perf_counter()-start_time)

print("Complete !")

best = results.max().round(1)
name_alg = ['Classic PSO','Improve PSO','Classic GA','Improve GA',
'GA other scheme','Improve PSO and GA','PSO and GA other scheme']
data = {'Algorithm': name_alg,
'C_best': (results[:,:,-1]>=best).sum(axis=1),
'Avg': (results[:,:,-1].sum(axis=1)/repeats).round(2),
'Med':np.median(results[:,:,-1],axis=1).round(2),
'Time':(time_work/repeats).round(2),
'Max':(results[:,:,-1].max(axis=1)).round(2),
'Min':(results[:,:,-1].min(axis=1)).round(2),}
df = pd.DataFrame(data=data)
print(df)
```

```
fig, ax = plt.subplots(figsize=(10,5))

fitness = results.sum(axis=1)/repeats
plot_colors = ['brown', 'green', 'orange', 'blue', 'red',
'purple', 'gray']
plot_markers = ["*",'o','^','s','p','D', "P"]

for i in range(1,7):
    ax.plot(range(n_iter+1),  fitness[i], color=plot_colors[i])

step = 40
start = 8
points = np.arange(0, n_iter+1, step)
for i in range(1,7):
    x = points+start*i
    x = x[x<=n_iter]
    ax.scatter(x,  fitness[i][x],color=plot_colors[i],
    label=name_alg[i], marker=plot_markers[i], s=100)
ax.legend()
ax.grid()
plt.xlabel('Iteration')
plt.ylabel('Average␣$f_{max}$')
plt.savefig(f"avg.png")
plt.show()

fig, ax = plt.subplots(figsize=(10,5))
for i in range(1,7):
    ax.plot(range(50),  fitness[i][:50], color=plot_colors[i])

step = 7
start = 1
points = np.arange(0, n_iter+1, step)
for i in range(1,7):
    x = points+start*i
    x = x[x<50]
    ax.scatter(x,  fitness[i][x],color=plot_colors[i],
    label=name_alg[i], marker=plot_markers[i], s=100)
ax.legend()
ax.grid()
plt.xlabel('Iteration')
plt.ylabel('Average␣$f_{max}$')
plt.savefig(f"avg_first_iter.png")
plt.show()

fig, ax = plt.subplots(figsize=(10,5))
for i in range(1,7):
    ax.plot(range(n_iter-50,n_iter+1),  fitness[i][-51:],
    color=plot_colors[i])

step = 7
start = 1
points = np.arange(0, n_iter+1, step)
for i in range(1,7):
    x = points+start*i
    x = x[x<=n_iter]
    x = x[x>=n_iter-50]

    ax.scatter(x,  fitness[i][x],color=plot_colors[i],
    label=name_alg[i], marker=plot_markers[i], s=100)
```

```
ax.legend()
ax.grid()
plt.xlabel('Iteration')
plt.ylabel('Average␣$f_{max}$')
plt.savefig(f"avg_last_iter.png")
plt.show()

step = 20
start = 7
points = np.arange(0, n_iter+1, step)
for j in range(7):
    fig, ax = plt.subplots(figsize=(5,5))
    for i in range(7):
        ax.plot(range(n_iter+1),  results[j,i],color=plot_colors[j])
    ax.plot(range(n_iter+1),  results[j,i],color=plot_colors[j],
    label=name_alg[j])
    ax.legend()
    ax.grid()
    plt.xlabel('Iteration')
    plt.ylabel('$f_{max}$')
    plt.savefig(f"lines/line{j}.png")
    plt.show()

bins = 16

for j in range(7):
    fig, ax = plt.subplots(figsize=(8,4))
    y = results[j,:,-1]
    step = (y.max() - y.min())/bins/2
    x = np.linspace(y.min(), y.max(), bins)
    density, _ = np.histogram(y, bins=bins)

    ax.bar(height = density, x=x, width = step*2, alpha=0.8,
    color=plot_colors[j],
    label=name_alg[j])
    ax.grid(True, axis='y')
    if j>3:
        plt.xticks(np.arange(y.min(),y.max()+0.1,0.5))
    ax.legend()
    plt.ylabel('Counts')
    plt.xlabel('$f_{max}$')
    plt.savefig(f"hists/hist{j}.png")
    plt.show()

scheme, fit = improve_gen_alg(shape, sizes_zones, count_pop,
count_kids, prob_mutate, max_mutations, prob_crossover,
function_for_array, corr_array, n_iter,
seed=seeds[results[3,:,-1].argmax()+repeats*3])
print(f"Scheme␣with␣value␣of␣objective␣function␣{round(fit[-1],2)}")
print(scheme[-1].reshape(shape))
```

# ДОДАТОК Б РЕЗУЛЬТАТИ ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ДЕТАЛЬНИЙ АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

У доданку запропоновані та наведені додаткові модифікації генетичного алгоритму та методу на основі рою частинок, що не увійшли в основну частину роботи. Було надано їх результати роботи разом з іншими методами у вигляді таблиць та рисунків, і на основі них було проведено більш детальний аналіз отриманих результатів.

## Б.1 Використані методи та їх модифікації

Для тестування обрані всі методи та їх модифікації, що згадані у розділі 3, а саме звичайний PSO, PSO з модифікованим відбором рішень, звичайний GA, GA з модифікованим відбором рішень та GA з модифікованою схемою представлення та обробки задачі. Також було реалізовано дві додаткові модифікації, а саме:

– *Комбінація PSO з модифікованим відбором та GA з модифікованим відбором.* Ця комбінація є поєднанням двох модифікацій, що виконуються по черзі, а саме спочатку на одну ітерацію запускається PSO з модифікованим відбором, і отримана на виході множина рішень ділиться на дві половини за значенням цільової функції. Краща половина рішень не змінюється, а гірша надається на вхід до GA з модифікованим відбором. Він також виконується один раз і отримані розв'язки замінюють ті, що подавались на вхід. Отримана множина рішень подається знов на вхід до PSO і ця процедура повторюється знов, поки не виконається критерій зупинки, тобто, у нашому випадку до досягнення максимальної кількості ітерацій.

– *Комбінація PSO з модифікованим відбором та GA з модифікованою схемою.* Ця комбінація також є поєднанням двох модифікацій, що виконуються по черзі, де на одну ітерацію запускається PSO з модифікованим відбором, і отримана на виході множина рішень

ділиться на дві половини за значенням цільової функції. Краща половина рішень не змінюється, а гірша видозмінюється, щоб відповідати новій схемі, та надається на вхід до GA з модифікованою схемою. Алгоритм виконується один раз і отримані розв'язки замінюють ті, що подавались на вхід. Отримана множина рішень подається знов на вхід до PSO і ця процедура повторюється поки не виконається критерій зупинки, а саме до досягнення максимальної кількості ітерацій.

## Б.2 Отримані результати тестування та їх аналіз

Щоб усереднити результат по досягненому значенню та по часу, тестування повторювалось 100 разів для кожного методу при однакових параметрах популяції. Кінцеві результати наведені в таблицях (табл. Б.1, Б.2), усереднені криві зміни знайденого найкращого значення цільової функції по ітераціях (рис. Б.3, Б.1, Б.2), приклади запусків алгоритмів (рис. Б.4, Б.5) та знайдені оптимальні схеми розміщення функціональних зон авіаційних логістичних парків (рис. 3.3) наведені на рисунках. Модель та методи були реалізовані за допомогою мови програмування Python, та їх написані тести наведені у додатку до роботи, а саме тестування проводилось у середовищі Google Colab на базі Intel(R) Xeon(R) CPU @ 2.20GHz Family 6 Model 79.

Порівнюючи результати між собою (табл. Б.1, Б.2), особливо на схемі $5 \times 5$, найчастіше найкращий результат показувала комбінація PSO й GA іншою схемою та відбором і GA з модифікованою схемою за більшістю параметрів: середнім результатом, медіаною, максимумом, та кількістю запусків, що знайшли глобальний максимум. Але також високі результати, особливо на схемі $3 \times 9$, також показали комбінація PSO й GA іншим відбором та GA з модифікованим відбором. Серед усіх адаптацій, лише PSO з модифікованим відбором працювала найгірше, і хоча продуктивність методу є кращою, ніж для класичний PSO, однак за різними параметрами вона є приблизно на рівні класичного GA.

**Таблиця Б.1** – Таблиця результатів оптимізації схеми розміщення авіаційного логістичного парку для схеми розміром $3 \times 9$

| Алгоритм | Розмір популяції | Кількість досягнень максимуму | Середнє | Медіана | Середній час роботи, с | Макс. | Мін. |
|---|---|---|---|---|---|---|---|
| Класичний PSO | 300 | 0 | 40,68 | 40,0 | 4,05 | 50,0 | 36,6 |
| PSO з іншим відбором | | 0 | 57,08 | 57,2 | 18,05 | 60,0 | 52,0 |
| Класичний GA | | 0 | 56,77 | 57,0 | 5,65 | 59,6 | 52,2 |
| GA з іншим відбором | | 3 | 58,99 | 59,2 | 7,03 | 60,2 | 56,4 |
| GA з іншою схемою | | 2 | 58,92 | 59,2 | 13,37 | 60,2 | 55,6 |
| PSO GA інший відбір | | 2 | 59,16 | 59,2 | 27,36 | 60,2 | 57,6 |
| PSO GA інша схема та відбір | | 2 | 59,15 | 59,2 | 33,01 | 60,2 | 56,8 |
| Класичний PSO | 600 | 0 | 42,81 | 42,5 | 6,68 | 49,4 | 36,6 |
| PSO з іншим відбором | | 0 | 57,80 | 57,8 | 32,61 | 59,8 | 53,0 |
| Класичний GA | | 1 | 57,42 | 57,8 | 10,39 | 60,2 | 52,8 |
| GA з іншим відбором | | 3 | 59,29 | 59,4 | 13,21 | 60,2 | 56,0 |
| GA з іншою схемою | | 5 | 59,34 | 59,4 | 24,36 | 60,2 | 57,2 |
| PSO GA інший відбір | | 3 | 59,16 | 59,2 | 48,60 | 60,2 | 57,6 |
| PSO GA інша схема та відбір | | 7 | 59,33 | 59,4 | 60,02 | 60,2 | 57,8 |

Найгірші показники, як за результатами (табл. Б.1, Б.2), так і за характером збіжності (рис. Б.3), окрім часу, має класичний PSO, оскільки цільова функція в цій задачі є складною та негладкою, та замість дослідження лише розв'язків, для яких виконувались встановлені обмеження на розміри площ функціональних зон, доводиться сканувати увесь простір можливих значень.

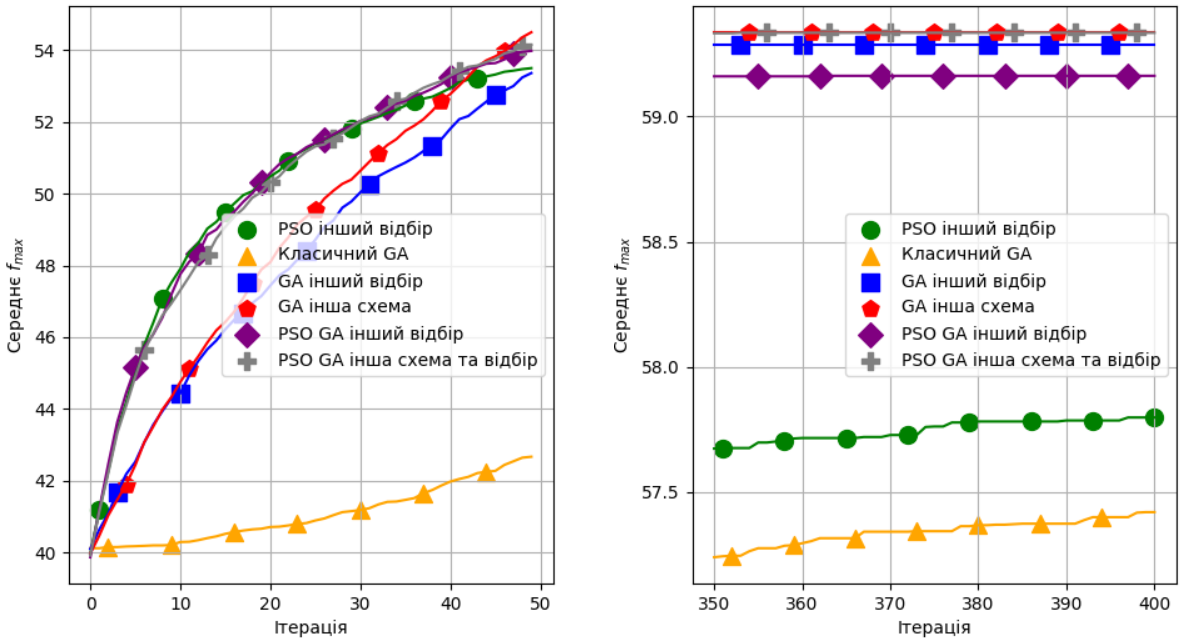
На графіках з досягнутого найкращого значення на ранніх та пізніх етапах (рис. Б.1, Б.2) видно, що PSO з модифікованим відбором та комбінації PSO й GA мали кращі результати для початкових 40 ітерацій, що свідчить про те, що адаптації та комбінації PSO мають високі здібності для вдосконалення початкових рішень. Однак вже наприкінці видно, що PSO з модифікованим відбором застряг в локальних оптимумах, тоді як адаптації та комбінації GA досягли високих результатів, що свідчить про їх кращу здатність до пошуку нових рішень. В той же час класичний GA показує слабку швидкість збіжності на

**Таблиця Б.2** – Таблиця результатів оптимізації схеми розміщення авіаційного логістичного парку для схеми розміром $5 \times 5$

| Алгоритм | Розмір популяції | Кількість досягнень максимуму | Середнє | Медіана | Середній час роботи, с | Макс. | Мін. |
|---|---|---|---|---|---|---|---|
| Класичний PSO | 300 | 0 | 39,51 | 38,8 | 4,17 | 47,2 | 34,6 |
| PSO з іншим відбором | | 1 | 51,59 | 51,6 | 17,96 | 54,2 | 49,0 |
| Класичний GA | | 1 | 52,01 | 52,0 | 5,59 | 54,2 | 49,4 |
| GA з іншим відбором | | 4 | 52,70 | 52,4 | 6,93 | 54,2 | 51,0 |
| GA з іншою схемою | | 8 | 53,01 | 53,2 | 12,73 | 54,2 | 51,0 |
| PSO GA інший відбір | | 10 | 52,89 | 53,0 | 26,70 | 54,2 | 51,0 |
| PSO GA інша схема та відбір | | 18 | 53,29 | 53,3 | 32,47 | 54,2 | 51,4 |
| Класичний PSO | 600 | 0 | 41,14 | 40,8 | 6,79 | 48,0 | 35,2 |
| PSO з іншим відбором | | 12 | 52,31 | 52,0 | 32,15 | 54,2 | 49,8 |
| Класичний GA | | 1 | 52,11 | 52,0 | 10,48 | 54,2 | 49,6 |
| GA з іншим відбором | | 6 | 52,92 | 53,0 | 13,49 | 54,2 | 51,0 |
| GA з іншою схемою | | 22 | 53,36 | 53,4 | 23,66 | 54,2 | 51,8 |
| PSO GA інший відбір | | 14 | 53,13 | 53,2 | 48,23 | 54,2 | 50,8 |
| PSO GA інша схема та відбір | | 21 | 53,42 | 53,3 | 58,72 | 54,2 | 52,2 |

ранніх ітераціях через суттєву різницю між хромосомами нові утворені рішення часто були неприйнятними, але з часом для всієї популяції вибудовується майбутній приблизний макет схеми, і під час цього хромосоми стають більш схожими одина на одну.

Аналізуючи швидкість збіжності з отриманим результатом (рис. Б.3, Б.4, Б.5), то хоча PSO з модифікованим відбором та класичний GA показують деякий ріст протягом всіх ітерацій, однак після 300 ітерацій він вже можливий лише завдяки випадково знайденій вдалій модифікацій після багаторазового перебору, тому подальший ріст є несуттєвим та вже навряд чи досягне нових значень. Водночас адаптації та комбінації GA вже отримали значні результати та завершують свою збіжність ще до $200 - 250$ ітерацій, тому для цих методів є допустимим зменшення максимальної кількості ітерацій до цього значення, і за рахунок цього можливе зменшення часу роботи цих алгоритмів приблизно в 2 рази без суттєвої втрати точності.




(а) (б)

**Рисунок Б.1** – Графік значень цільової функції по ітераціям для різних алгоритмів в середньому після 100 запусків для схеми розміром $3 \times 9$: (а) для перших 50 ітерацій, (б) для останніх 50 ітерацій

Щодо часу роботи алгоритмів (табл. Б.1, Б.2), то найбільш швидкими методами є класичний GA та PSO завдяки тому, що вони є найбільш простими та потребують мінімум обчислень. Якщо їх порівнювати з модифікаціями GA та PSO іншого відбору, то видно, що вдосконалення GA витрачає значно менше часу в порівнянні з вдосконаленням PSO. Це пов'язане з тим, що коли популяція GA вже частково або повністю збіглась, то через схожість хромосом поставлені обмеження на розміри площ будуть частіше виконуватись при утворенні нових рішень, і впроваджена модифікація буде рідше викликатись. Водночас популяція PSO постійно подорожує в просторі рішень, навіть коли метод збігся до певного оптимуму, через що необхідно постійно робити додаткові розрахунки та здійсьнювати пошук прийнятних рішень, що суттєво впливає на кінцевий результат роботи. Схожим чином GA з

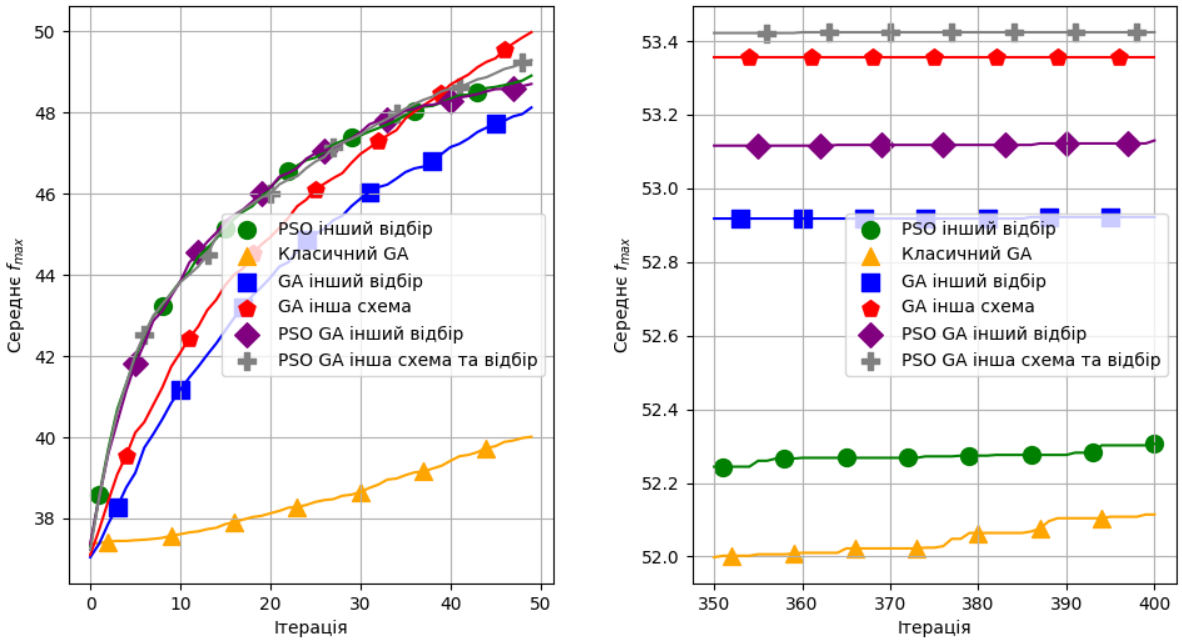
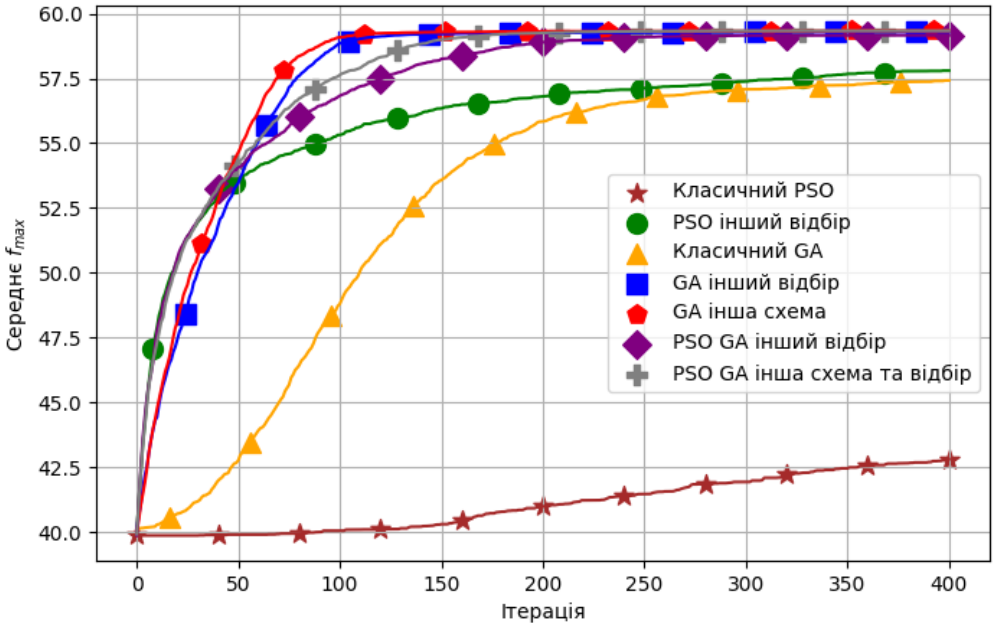
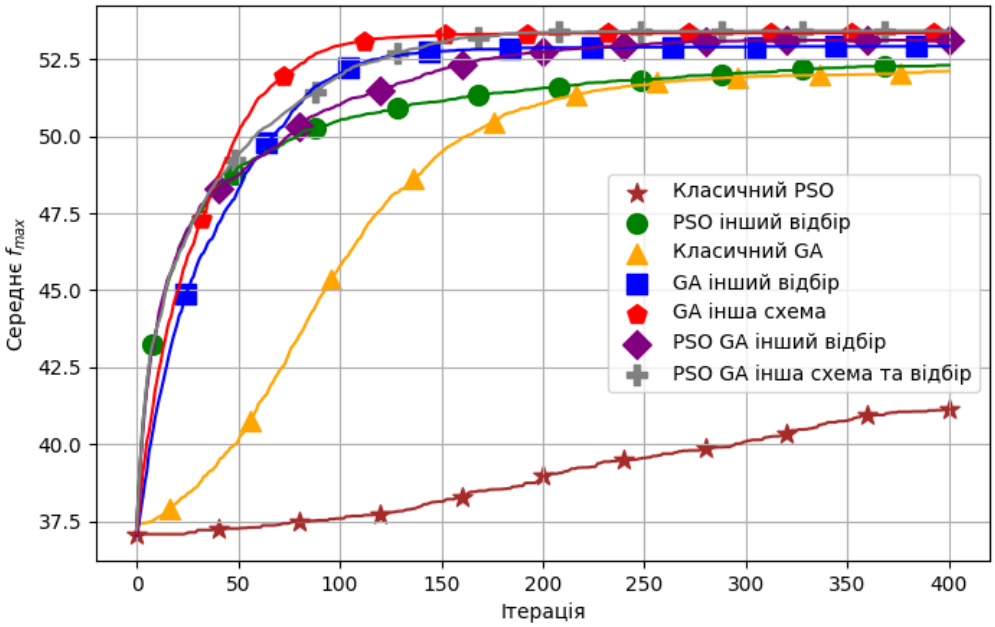



**Рисунок Б.2** – Графік значень цільової функції по ітераціям для різних алгоритмів в середньому після 100 запусків для схеми розміром $5 \times 5$: (а) для перших 50 ітерацій, (б) для останніх 50 ітерацій

модифікованою схемою витрачає певний час, щоб для частини отриманих хромосом прибрати дублікати номерів та розмістити втрачені значення. Найдовше серед усіх методів працювали комбінації PSO й GA, бо вони фактично викликають обидва методи на кожній ітерації, і отриманий час роботи рівний сумі тривалості виконання цих методів окремо разом з додатковими затратами, що, наприклад, можуть виникати через постійні зміни схем представлення задачі.

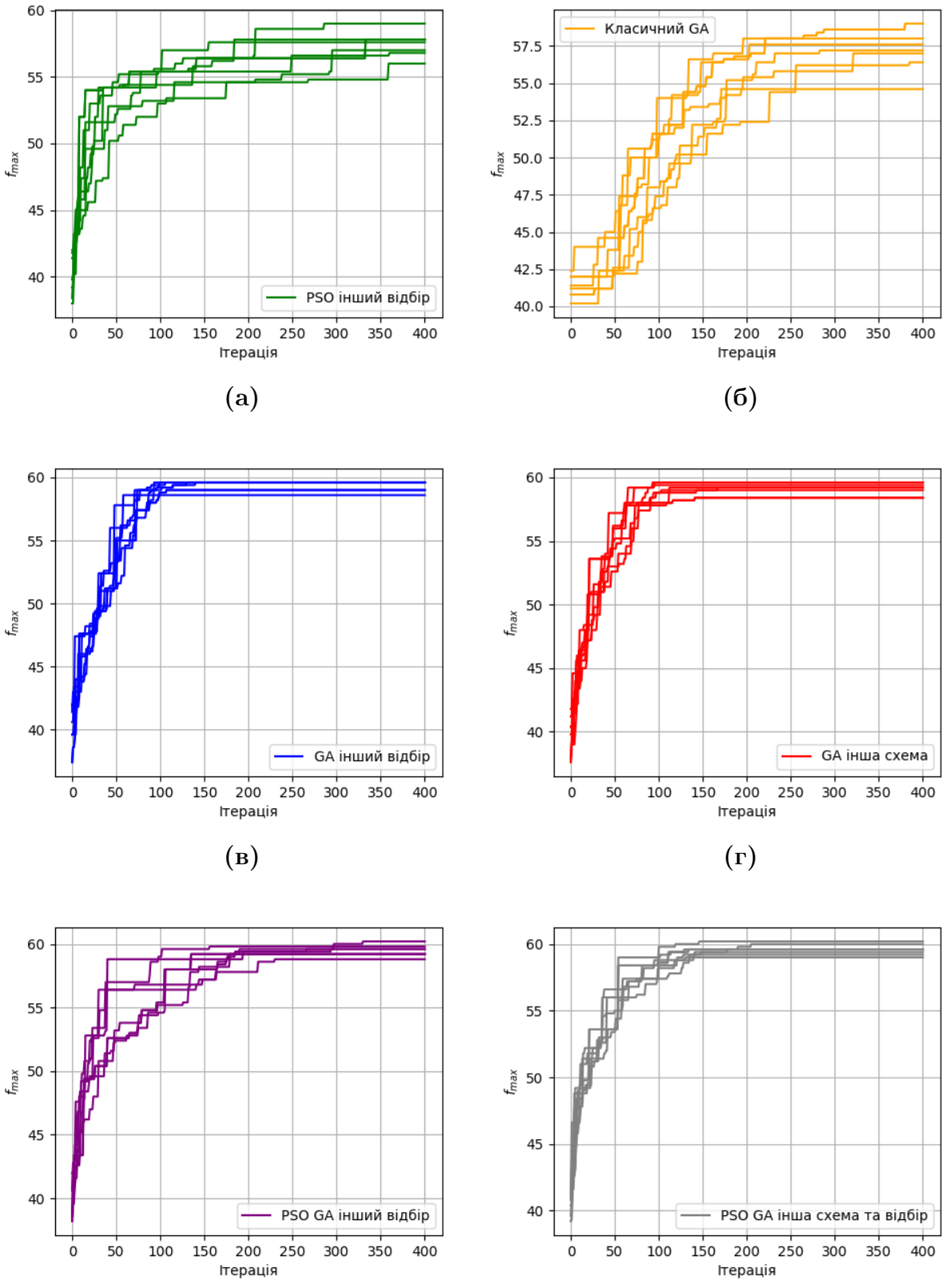



(а)



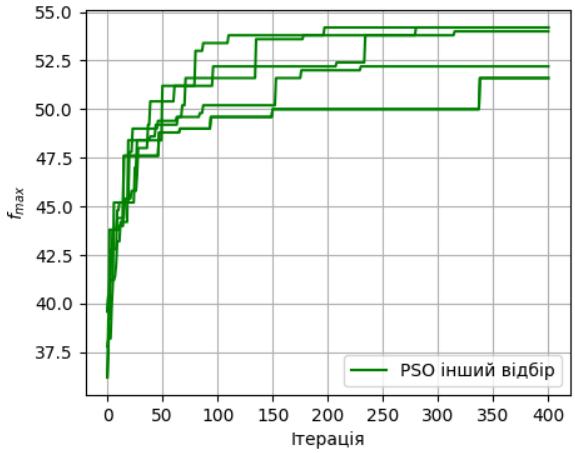

(б)

**Рисунок Б.3** – Графік досягнутих значень цільової функції по ітераціям для різних алгоритмів в середньому після 100 запусків: (а) для схеми розміром $3 \times 9$, (б) для схеми розміром $5 \times 5$

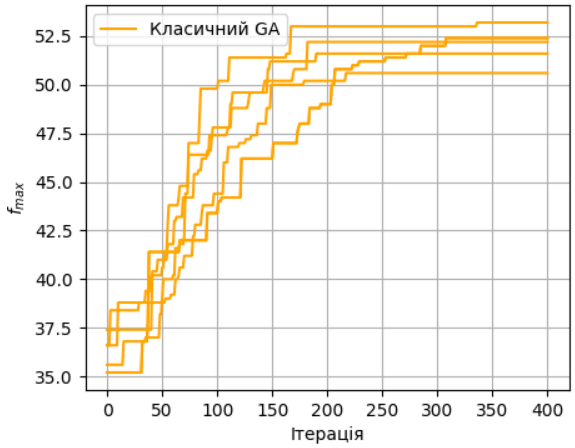



(а) (б) (в) (г) (д) (е)

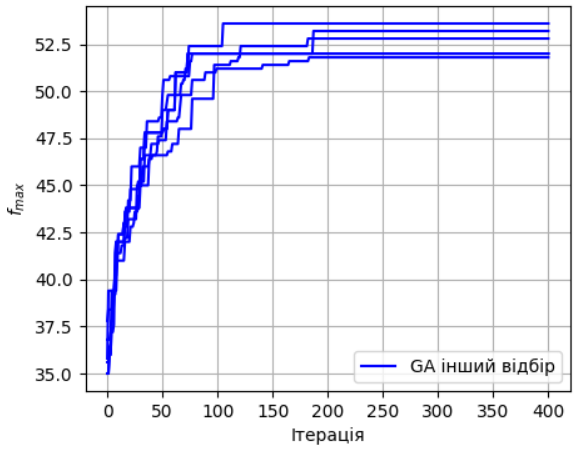
**Рисунок Б.4** – Приклад кількох запусків для різних алгоритмів для схеми $3 \times 9$



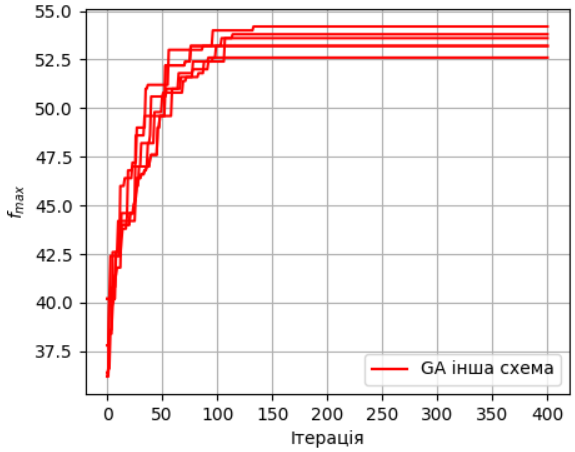

(а)



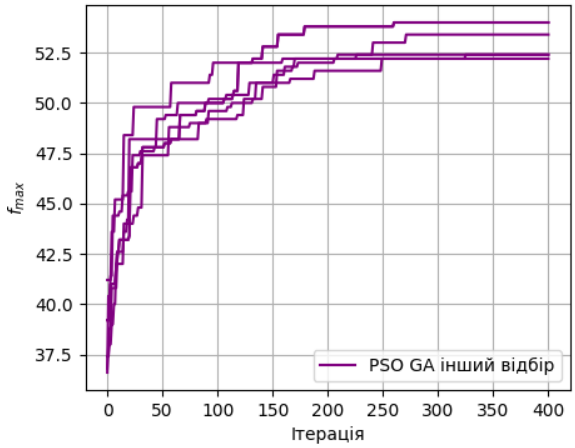

(б)



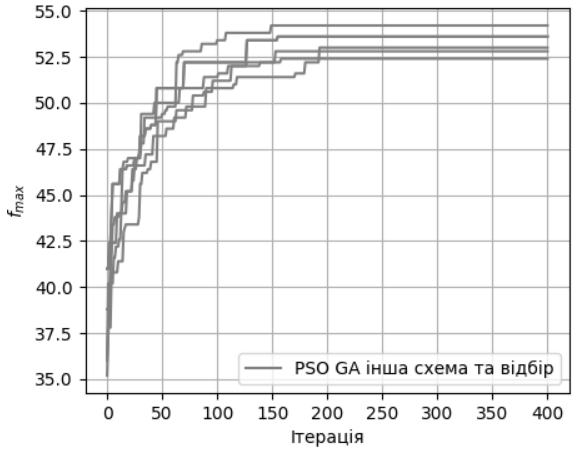

(в)




(г)




(д)




(е)

**Рисунок Б.5** – Приклад кількох запусків для різних алгоритмів для схеми $5 \times 5$