МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ
ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ Аліна САВЧЕНКО
«___»_____2022 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТР
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

Тема: «Система реалізації задачі дискретної оптимізації з використанням еволюційного алгоритму»

Виконавець: Олександр БАБІЙЧУК

Керівник: к.пед.н., доцент Юрій СІНЬКО

Нормоконтролер: к.т.н., доцент Олена ТОЛСТІКОВА

КИЇВ 2022

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *кібербезпеки, комп'ютерної та програмної інженерії*
Кафедра *Комп'ютерних інформаційних технологій*
Спеціальність *122 «Комп'ютерні науки»*
Освітньо-професійна програма *«Інформаційні технології проектування»*

ЗАТВЕРДЖУЮ:
завідувач кафедри КІТ
_____Аліна САВЧЕНКО
(підпис)
«____»_____2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Бабійчука Олександра Юрійовича
(ПІБ випускника)

1. Тема роботи: «Система реалізації задачі дискретної оптимізації з використанням еволюційного алгоритму № 1774/ст від 28.09.2022р.
2. Термін виконання роботи: з 26 вересня 2022 року по 21 листопада 2022 року.
3. Вихідні дані до роботи: Способи реалізації задачі дискретної оптимізації з використанням еволюційного алгоритму.
4. Зміст пояснювальної записки: 1. Огляд та аналіз предметної області. 2. Огляд алгоритмів та технологій для розробки системи. 3. Розробка та тестування системи.
5. Перелік обов'язкового ілюстративного матеріалу: 1. Актуальність та практична цінність теми. 2. Об'єкт дослідження. 3. Мета роботи. 4. Задачі кваліфікаційної роботи. 5. Демонстрація системи. 6. Висновки.

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Аналіз предметної області та огляд аналогів. Написання 1 розділу, представлення керівнику.	26.09.2022-16.10.2022	
2.	Вибір та опис використаних технологій. Написання 2 розділу, представлення керівнику.	17.10.2022-30.10.2022	
3.	Розробка додатка за допомогою обраного стеку технологій. Написання 3 розділу, представлення керівнику.	31.10.2022-14.11.2022	
4.	Загальне редагування та друк пояснювальної записки.	15.11.2022-20.11.2022	
5.	Проходження нормоконтролю, перепліт пояснювальної записки.	16.11.2022-20.10.2022	
6.	Розробка тексту доповіді. Оформлення графічного матеріалу для презентації.	20.11.2022-22.11.2022	

7. Дата видачі завдання 26.09.2022р.

Керівник кваліфікаційної роботи _____ Юрій СІНЬКО
(підпис керівника)

Завдання прийняв до виконання _____ Олександр БАБІЙЧУК
(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Система реалізації задачі дискретної оптимізації з використанням еволюційного алгоритму» містить: 84 сторінки, 25 рисунків, 1 таблицю, 20 інформаційних джерел.

Об’єкт дослідження – система реалізації задачі дискретної оптимізації з використанням еволюційного алгоритму.

Предмет дослідження – використанням еволюційного алгоритму для вирішення задачі дискретної оптимізації.

Мета кваліфікаційної роботи – дослідити методи розробки систем оптимізації та алгоритми що в них використовуються. Розробитисистему для вирішення задачі дискретної оптимізації на основі еволюційного алгоритму.

Методи дослідження – аналііз наукових робіт та технічної документації, мова програмування *Java*, середовище розробки *IntelliJ IDEA*.

Результати магістерської роботи рекомендується використовувати для розробки систем оптимізації з використанням мови програмування *Java*, методології розробки програмних ghjlernsd, системи управління проектами.

ДИСКРЕТНА ОПТИМІЗАЦІЯ, ЕВОЛЮЦІЙНИЙ АЛГОРИТМ, ГЕНЕТИЧНИЙ АЛГОРИТМ, *JAVA CORE*.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

АКО	—	Алгоритми комбінаторної оптимізації
ЗДО	—	Задачі дискретної оптимізації
МГіМ	—	Метод гілок і меж
ПАВ	—	Послідовний аналіз варіантів
GA *(Genetic Algorithm)*	—	Генетичний алгоритм
TS *(Tabu search)*	—	Табу-пошук
CS (Cuckoo Search)	—	Зозулин пошук
JDK (Java Development Kit)	—	Безкоштовний розповсюджуваний Oracle комплект розробника застосунків на мові Java.
GUI (graphical user interface)	—	Графі́чний інтерфе́йс кори́стувача́
ООП	—	Об'єктно-орієнтоване програмування
URL *(Uniform Resource Locator)*	—	Визначник місцезнаходження сайту в мережі Інтернет
IT *(Informational technologies)*	—	Інформаційні технології

ВСТУП

Вивчення питань прийняття рішень під час дослідження й проектування складних систем призводить до необхідності постановок задач оптимізації з урахуванням сукупності критеріальних функцій — багатокритеріальних задач оптимізації. Це пояснюється тим, що лише у виняткових випадках альтернативні рішення щодо способів проведення операції можна порівнювати між собою за одним критерієм. Зазвичай для прийняття рішення потрібно здійснити вибір на основі цілої низки критеріїв, які можуть перебувати в суперечності один до одного. Але при цьому кожен із запропонованих критеріїв вважається настільки суттєвим, що не врахування його під час вибору рішення було б ризикованим і необачливим з огляду наслідків проведення операції. Основним питанням, що виникає, коли порівнюють між собою альтернативні рішення за умови кількох критеріїв, є: яке з двох рішень вважати кращим, якщо під час заміни одного із цих рішень на інше значення одного або кількох критеріїв «покращаться», а інших — «погіршаться». [1]

Оскільки в більшості подібних питань існує скінченна кількість рішень, то подібні питання є логічним віднести до задачі дискретної оптимізації. Це обумовлено тим, що дискретні оптимізаційні моделі адекватно відбивають нелінійні залежності та враховують обмеження логічного і технологічного типу, а також мають якісний характер. В даний час серед найбільш перспективних напрямків досліджень в області дискретної оптимізації можна виділити такі підходи:

- розробка ефективних обчислювальних алгоритмів (точних та наближених) для вирішення завдань дискретної оптимізації;
- пошук спеціальних класів задач дискретної оптимізації, на яких добре працюють ті чи інші алгоритми;
- розробка та дослідження алгоритмів дискретної оптимізації з ефективним розпаралелюванням обчислень;
- теоретичний аналіз складності алгоритмів розв’язання задач дискретної оптимізації.

Актуальність теми кваліфікаційної роботи «Система реалізації задачі дискретної оптимізації з використанням еволюційного алгоритму» ґрунтується на тому, що у наш час оптимізація є важливою і незамінною частиною кожного робочого процесу. Це зумовлено тим, що результатом оптимізації є знаходження більш ефективних шляхів виконання поставленої задачі, що дозволяє як для підвищити швидквсть виконання роботи, так і підвищити якімть кінцевого продукту.

Метою кваліфікаційної роботи є дослідження і розробка систем реалізації задачі дискретної оптимізації з використанням еволюційного алгоритму

Відповідно до поставленої мети роботи визначено основні **завдання дослідження**:

– провести аналіз наукової та методичної літератури;

– проаналізувати наявні способи вирішення задачі дискретної оптимізації;

– провести аналіз існуючих алгоритмів придатних для проведення дискретної оптимізації;

– розробити систему що здатна проводити дискретну оптимізацію;

– описати принцип роботи системи та її основних частин та провести тестування.

Для досягнення поставлених цілей і виконання поставлених завдань прийнято метод системного аналізу наукової літератури, який дає змогу виділити найважливіші елементи дискретної оптимізації та еволюційних алгоритмів.

Наукова новизна роботи включає в себе виділення особливостей роботи з еволюційними алгоритмами та використання їх для проведення дискретної оптимізації.

Практичною цінністю роботи виступає можливість використання розробленої системи для формування початкового розкладу, або як прклад реалізації еволюційного алгоритму для аналогічних систем оптимізації.

РОЗДІЛ 1. ОГЛЯД ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Визначення задачі дискретної оптимізації

Задачі дискретної оптимізації — задачі знаходження екстремумів заданої функції на дискретній (зазвичай скінченній) множині точок. Якщо функціональна область складається з кінцевої кількості точок, то дискретні задачі оптимізації в принципі завжди можна розв'язати шляхом обходу всієї множини. Однак на практиці ця колекція зазвичай дуже велика, тому методи сортування не працюють добре.

Для кращого розуміння розберемо це поняття докладніше.

Через якісне питання (або просте запитання) ми зрозуміємо щось загальнеп питання, на яке потрібно відповісти. Зазвичай завдання включають декілька параметр або вільна змінна, конкретне значення якої не визначено. Задача P визначається такою інформацією: загальний перелік усіх його параметрів, формула для властивостей, яким він повинен задовольняти

Відповідь, іншими словами, рішення проблеми.Індивідуальна задача *I* виходить з масової П, якщо всім параметрам задачі П присвоїти конкретні значення. Іншими словами, індивідуальна задача оптимізації – це пара *(F, C),* де *F* – довільна множина, область допустимих точок, а *C* – функція вартості, яка здійснює відображення $C: F \to R$. Потрібно знайти точку $f \in F$ для якої $C(f) < C(y)$ для всіх $y \in F$. Така точка називається глобальним оптимальним рішенням.

Для всіх (для деяких) задач математичного програмування дозволяють компонентам параметра відповідати цілочисловим вимогам, називається цілочисельною (частково цілочисельною) проблемою програмування.

Задачі цілочисельного програмування, які дозволяють тільки компонентам векторних альтернатив 0 або 1 приймати значення, називаються задачами цілочисельного програмування з булевими змінними або задачами булевого (бівалентного) програмування.

Задачі цілочисельного програмування, в яких допустимі лише значення компонент векторних альтернатив $\overline{(0, k-1)}$, називаються задачами програмування з *k*-значними змінними або *k*- значного програмування (у загальному випадку число *k* є відмінним для різних компонент).

Задача оптимізації скінченної множини об'єктів, побудованих за певними комбінаторними правилами, називається задачею комбінаторного програмування.

Усі перераховані задачі оптимізації вирішуються на відповідних Дискретні множини (у більшості випадків – скінченні). Загалом, дискретна множина — це множина, точки якої є ізольованими точками цієї множини.

В принципі, кожна задача комбінаторного програмування може бути Подається у вигляді задачі цілочисельного (частково цілого) програмування. Однак такий образ не завжди доречний.

Таким чином, оптимізаційна задача - це набір I окремих задач оптимізація. В одній задачі надаються вхідні дані та достатньо інформації для досягнення рішення, тоді як проблема оптимізації — це набір окремих проблем, створених однаковим чином. Отже, якщо мова йде про індивідуальні завдання комівояжера, то дана матриця відстаней. Якщо ми розглядаємо задачу комівояжера в цілому, ми маємо на увазі сукупність усіх окремих задач, що відповідають усім матрицям відстаней.

Оскільки в розрахунку буде розглядатися задача оптимізації аспекти, необхідно позначити зображення для однієї комбінованої проблеми

Оптимізуйте введення з комп'ютера. Можна, звичайно, виписати всі допустимі рішення і для кожного вказати значення *C*. Проте для більшості задач деякі індивідуальні задачі матимуть непропорційно велике число допустимих рішень, як це має місце, наприклад, в задачі комівояжера. Надалі

припускатимемо, що F і C задані неявно за допомогою двох алгоритмів A^* і A^{**}. Алгоритм A^* щодо даного комбінаторного об'єкта f і множини параметрів S вирішує, чи є f множиною рішень F, які визначаються даними параметрами. У свою чергу A^{**} за даним допустимим рішенням f і іншою множиною параметрів Q видає значення $C(f)$.

Тоді індивідуальну комбінаторну задачу I можна визначити, як зображення параметрів S і Q з використанням фіксованого кінцевого алфавіту і деякого стандартного розумного кодування.

Отже, комбінаторна оптимізаційна задача П – це обчислювальна задач а, що складається з трьох частин, а саме з: множини D_n індивідуальних задач; допустимих рішень індивідуальної задачі I для кожної $I \in D_n$ кінцевої множини $S_n(I)$; функції m_n, яка зіставляє кожній індивідуальній задачі $I \in D_п$ і кожному допустимому рішенню $\sigma \in Sn(I)$ деяке позитивне ціле число, назване величиною рішення σ.

Інакше кажучи, за даними зображення параметрів S і Q для алгоритмів A^* і A^{**} знаходження оптимального допустимого рішення називають оптимізаційним варіантом даної задачі.

Якщо необхідно за даними S і Q знайти вартість оптимального рішення, то такий варіант називають обчислювальним варіантом комбінаторної задачі оптимізації. Якщо вартість C треба просто обчислити, то обчислювальний варіант комбінаторної оптимізації не може бути набагато складнішим, ніж оптимізаційний варіант.

Особливо важливим при вивченні обчислювальної складності задачі є третій варіант комбінаторної задачі оптимізації. Цей варіант, названий варіантом розпізнавання, має наступний вигляд. Для даної індивідуальної задачі I, тобто представлення S і Q і цілого числа L, визначити, чи існує таке допустиме рішення F, що $C(f) < L$.

У відмінності від двох раніше розглянутих варіантів, варіант розпізнавання є питанням, на який можна відповісти так чи ні. Очевидно, відповісти на це питання не набагато важче, ніж розв'язати відповідну

обчислювальну задачу, оскільки після її розв'язання залишається тільки порівняти оптимальну вартість *C(f)* з *L* і видати відповідь так, щоб *C(f) < L*.

Тому використовувати лише припущення C(f) легко для розрахунку визначення кожного варіанту - оптимізації, розрахунку та ідентифікації - не складніше попереднього. З'являється питання в тому, чи не однакові варіанти за рівнем складності? Іншими словами, чи можна ефективно звертатися до обчислювальних варіантів за допомогою гіпотетичні алгоритми для вирішення проблеми визначення варіантів і чи можуть варіанти оптимізації та розрахунку зробити те саме?

Якщо вартість оптимального рішення є цілим числом, алгоритм якого обмежений поліномом від розміру вхідних даних, то доки варіант може ефективно вирішити проблему ідентифікації, а також застосовується до варіантів розрахунку.

Слід зазначити, що справедливість припущення log C(f),м омежується поліномом розміру вхідних даних для рішення обчислювальної задачі, будь-який алгоритм, який вирішує задачу розпізнавання, можна ефективно використовувати. Немає відомого загального методу вирішення оптимізованого варіанту задачі за допомогою алгоритму, який обчислює варіанти.

Багато задач оптимізації допускаються і можуть бути представлені графічно ефективно розв'язується методами теорії графів. Ці проблеми добре описані мовою дискретного програмування і, таким чином, можуть розглядатися як частина проблеми в цьому напрямку. Враховуйте специфіку завдання, і їх формулювання на мові теорії графів допускає більш прості методи рішення задачі на порядок більше, ніж розмірність цієї ж задачі в загальній постановці дискретного програмування.

Графічна інтерпретація теорії, навпаки, забезпечує наочність і прийнятність для постановки досить складних прикладних задач і способів їх вирішення.

Значення задач цілочисельної та комбінаторної оптимізації стрімко зростає. Це пов'язано з необхідністю пошуку ефективних рішень технічних, технічних, організаційних і соціальних проблем, які з часом ускладнюються. У багатьох випадках такі рішення після відповідної формалізації зводяться до задач вибору на скінченних множинах або задач змішаного вибору на скінченних і неперервних множинах.

Перелік прикладних задач цілочисельного і комбінаторного програмування доволі різноманітний.

До числа основних типів задач, що здобули найширшу популярність, відносяться задачі:

- про рюкзак - визначити оптимальну сукупність неподільних предметів, які задовольняють обмеженням об'єму або (і) ваги, або (і) енергоспоживання і т. п.;
- про мінімальні покриття при застосуванні до синтезу технічних пристроїв;
- про максимальне інцидентне поєднання при застосуванні до формування організаційних систем;
- щодо розподілу - в ширшому розумінні оптимальний розподіл завдань між виконавцями - склад і розподіл завдань через звязку засобів оптимізації організаційно-технічного комплексу;
- про розміщення – оптимальне розташування та потужність джерел у мережі; пошук оптимальних шляхів у мережі, особливо навколо n точок (завдання комівояжера), оптимальне планування організації потоку процесів, процесу обслуговування, навчання та ін.[2]

1.2. Методи розв'язування задач дискретної оптимізації

Методи розв'язування задач дискретної оптимізації поділяють на дві основні підгрупи:

- аналітичні;
- алгоритмічні.

Аналітичні методи є методами які людина зазвичай використовує без додаткового обладнання і спеціальних знань. Основною функцією даних методів є відсікання неможливих та найбільш очевидних неефективних варіантів.

Алгоритмічні методи базуються на побудові алгоритму в результаті якого і буде отриманий результат. Такі алгоритми зазвичай будуються з використанням правил комбінаторики, тому їх і звуть комбінаторні алгоритми. Комбінаторні алгоритми це доволі широке поняття, тому вони мають власну класифікацію.

Алгоритми комбінаторної оптимізації класифікують за багатьма характеристиками - за точністю, типом використаних просторів, структурою обчислювальної схеми тощо (рис. 1).

Рис. 1.1. Класифікація АКО за точністю

Точні АКО - це такі методи, які знаходять глобальний розв'язок. Наближений тип АКО поділяється на алгоритми з апріорною та апостериорною оцінками точності. Евристичні алгоритми засновані на обґрунтованому міркуванні (наприклад, алгоритм «їдь до найближчого міста» для вирішення задачі комівояжера), але не можуть оцінити точність знайдених рішень. Зазвичай алгоритми оцінки точності та евристичні алгоритми дослідники називають алгоритмами апроксимації, тому далі ми розглянемо комбінації таких прикладних алгоритмів.

Класифікація АКО за типом обчислювальної схеми.

Наближені алгоритми за типом обчислювальної схеми прийнято ділити на конструктивні та ітераційні. Нехай маємо певну множину $Y \supseteq X$. Конструктивні алгоритми (інші назви - прямі, послідовні) - це такі алгоритми, у яких $Y \supset X (Y \neq X)$, тобто вони оперують у просторі, що є розширенням простору розв'язків *X* Починаючи "з нуля" чи якогось фрагмента розв'язку, вони поступово формують повний розв'язок.

Приклади конструктивних алгоритмів для різних ЗДО:

1) Задача комівояжера - алгоритм "іди в найближче місто", який на кожному наступному кроці додає до маршруту вершину, відстань до якої є найменшою з усіх можливих.

2) Квадратична задача про призначення - евристика, відповідно до якої об'єкти (елементи) з інтенсивнішими потоками розміщуються в першу чергу.

3) Мінімальне кістякове дерево - евристика, відповідно до якої на кожному наступному кроці в побудований фрагмент кістякового дерева включається та вершина, яка мінімально збільшує його сумарну довжину й не утворює підцикл.

Ітераційні алгоритми - це такі алгоритми, які на кожному кроці опрацьовують "повні" розв'язки, тобто для них простір пошуку *Y= X* Починаючи з деякого $x^\circ \in X$, ітераційні алгоритми намагаються його поліпшувати покроково:

$$x^{(h+1)} = A^{(h)} x^{(h)}, \; h = 0,1,\ldots,$$

де $A^{(h)}$ - ітераційна процедура, яка у більшості випадків не залежить від кроку *h*, тобто $A^{(h)} = A$.

Ітераційні методи, які працюють з (поточним) розв’язком на кожному кроці, називаються траєкторними методами; інколи в зарубіжній літературі такі методи називають *Single-Solution Based/Single-State Methods*, а траєкторні методи відносяться до свого такого підкласу, який генерує послідовність суміжних рішень – траєкторій у просторі пошуку.

Щоб уникнути деяких термінологічних ускладнень, ми називаємо всі такі методи траєкторіями. Алгоритми, які одночасно обчислюють кілька рішень замість одного на кожній ітерації, називаються методами на основі сукупності.Отже, для траєкторних алгоритмів $||Z|| = || X^* || = 1$, а для популяційних - $||Z|| > 1$, $||X^*|| > 1$.

За складністю структури АКО можна виділити:

- прості алгоритми;
- комбіновані алгоритми;
- метаевристики;
- гібридні метаевристики;
- гіперевристики.

Комбінаторні алгоритми утворюються шляхом послідовного застосування двох або більше ітераційних алгоритмів і перенесення рішення від одного алгоритму до іншого. У метаевристики, яка буде обговорюватися пізніше, один алгоритм/процедура вбудована в іншу стратегію. Гіперевристичний алгоритм — це метод пошуку, який зосереджується на процесі автоматичного вибору, комбінування або адаптації або налаштування кількох простіших алгоритмів (евристик або метаевристик) для отримання ефективного рішення для задач дискретної оптимізації або його класу. Цього можна досягти шляхом вибору існуючих евристик або їх фрагментів і створення нових евристик.

Отже, якщо метаевристики та інші алгоритми в основному шукають у просторі розв’язків задач дискретної оптимізації, то простір пошуку гіперевристик – це набір евристик (простіших алгоритмів або їх частин).[3]

1.3. Точні алгоритми

Точні алгоритми можна поділити на загальні методи та спеціальні алгоритми.[3]

Загальні методи:

- повний перебір (вичерпний пошук);

- метод гілок і меж (МГіМ);
- метод гілок і відтинань;
- послідовний аналіз варіантів ("київський віник");
- динамічне програмування (метод Беллмана).

Спеціальні алгоритми будуються на основі врахування специфіки конкретної задачі оптимізації, що розв'язується, тому мають вузький спектр застосування. Приклад - метод Балаша (угорський метод) для розв'язання лінійної задачі про призначення. Один з небагатьох прикладів поліноміального алгоритму для розв'язання ЗДО. Однак при додаванні однієї чи кількох обмежувальних умов виникають подібні задачі, але зазначений алгоритм уже не може бути застосованим чи його модифікація уже не має поліноміальної складності.

Розглянемо основні точні алгоритми. Повний перебір це просто перебір всих варіантів, він є найбільш не ефективним і дуже рідко використовується.

Метод гілок і меж є одним із комбінаторних методів. Він працює як для повних, так і для неповних цілих завдань. Його суть полягає в тому, щоб вибрати впорядкований варіант і розглянути лише ті з них, які корисні для пошуку оптимального рішення за деякими ознаками. Відповідно до загальної ідеї методу, на першому кроці задача розв’язується як задача лінійного програмування, тобто цілочисельна умова не розглядається. Якщо ви отримуєте оптимальний цілочисельний розв’язок задачі лінійного програмування, то це також є розв’язком цілочисельної задачі лінійного програмування. Якщо цілочисельний розв’язок недоступний, то xr представляє невід’ємну частину змінної x_r*, значення якої є дробом в оптимальному розв’язку задачі лінійного програмування. Після чого, інтервал $[x_r^*] < x_r < [x_r^*]+1$ виключається з розгляду, як такий, що не містить допустимих цілочисельних компонент розв’язку. Тому допустиме ціле значення x_r повинно задовольняти одну з нерівностей $x_r \le [x_r^*]$ або $x_r \ge [x_r^*]+1$.

Додавши кожну з цих умов до розв’язуваної задачі, ми отримаємо дві не пов’язані між собою задачі. У цьому випадку вихідне запитання називається

розбитим (розгалуженим) на два підпитання. Потім кожна підзадача розв’язується як задача лінійного програмування. Якщо для цілочисельної задачі прийнятний оптимальний розв’язок, то такий розв’язок слід записати як найкращий. При цьому немає потреби продовжувати розгалуження на підзадачі, оскільки вдосконалити отримане рішення, очевидно, неможливо. В іншому випадку підзадачу слід знову розділити на дві підзадачі, знову враховуючи умови. Звичайно, коли ми отримуємо прийнятне цілочисельне рішення однієї з підзадач, яке краще за попередню, воно фіксується, а не раніше.

Процес розгалуження триває доти, доки кожна підпроблема не приведе до цілочисельного розв’язку, або поки не буде визначено, що фіксоване розв’язання не можна покращити. У цьому випадку оптимальним буде фіксоване допустиме рішення.[4]

Алгоритм Беллмана-Форда — це алгоритм для обчислення найкоротших шляхів від вихідної вершини до всіх інших вершин у зваженому орієнтованому графі. Звичайно, він повільніший, ніж алгоритм Дейкстри для тієї ж задачі, але він є більш загальним, оскільки він може обробляти графіки з від’ємними вагами для деяких ребер. Алгоритм часто називають на честь двох його розробників, Річарда Беллмана та Лестера Форда, які опублікували його в 1958 та 1956 роках відповідно. Однак Едвард Форрест Мур також опублікував цей алгоритм у 1957 році, тому його іноді називають алгоритмом Беллмана-Форда-Мура.

Як згадувалося вище, алгоритм Беллмана-Форда працює на графах з негативно зваженими ребрами. Однак, якщо граф містить «негативний цикл», тобто цикл, у якому сума ваг ребер від’ємна, то для даного графа не існує дерева найкоротших шляхів (будь-який шлях цього типу можна покращити за допомогою кількох проходів через край утворюють негативний цикл). У цьому випадку алгоритм Беллмана-Форда може виявити цикли негативної довжини та повідомити про їх наявність, але він не може дати правильну

відповідь, що він не може знайти найкоротший шлях, якщо негативний цикл доступний з вершини джерела.

Напишемо алгоритм Беллмана-Форда докладніше. Для цього розглянемо деякий орієнтований граф зі зваженими ребрами, який не містить циклів від'ємної довжини. І припустимо, що вам потрібно знайти найкоротший шлях від вибраної вершини до всіх інших вершин даного графа:

1) Перед початком виконання алгорітму всі вершини графа вважаються непройденими, а ребра – не переглянутими. Кожній вершині в ході виконання алгоритму присвоюється число d_x, рівне довжині найкоротшого шляху з вершини a у вершину x, що включає тільки пройдені вершини. На першому кроці покладаємо $d_a = 0$ і $d_x = \infty$ для всіх x, відмінних від a. Також, на даному кроці, вершині a присвоюється мітка *«пройдена»* і покладається $y = a$ (y – остання з пройдених вершин).

2) Для кожної з вершин графа G наступним чином перерахувати величину d_x: $d_x = \min\{d_x, d_y + m_{yx}\}$ (1) (де m_{yx} – вага ребра (y,x)). Якщо $d_x = \infty$ для всіх непройдених вершин x, процедуру алгоритму Беллмана-Форда необхідно завершити (у вихідному графі відсутні шляхи з вершини a у непройдені вершини). В іншому випадку, мітку *«пройдена»* необхідно присвоїти тій з вершин x, для якої величина d_x є найменшою. Крім того, ребро, що веде в обрану на даному етапі вершину x вважається переглянутим (для цієї дуги досягався мінімум відповідно до виразу *(1)*). Після цього, поклавши $y = x$, ітераційний процес продовжується далі. Відмітимо, що якщо для деякої пройденої вершини x відбувається зменшення величини d_x, то з цієї вершини і инцидентного їй переглянутого ребра відповідні мітки знімаються.

3) Процедура алгоритму Беллмана-Форда завершується тільки тоді, коли всі вершини графа G пройдені і коли після чергового виконання кроку номер два жодне з чисел d_x не змінило свого значення.[5]

1.4. Неточні алгоритми

Необхідність розробки ефективних наближених АКО, які застосовуються у переважній більшості випадків на практиці, визначається низкою обставин:

- практично всі важливі задачі належать до NP-складних, тож точне їх розв'язання дуже проблематичне навіть із використанням сучасних і перспективних комп'ютерів;
- їх цільові функції мають зазвичай велику кількість локальних екстремумів;
- у багатьох прикладних проблемах дані задаються з певними похибками, що робить недоцільними ті істотні обчислювальні затрати, які необхідні для знаходження їх точного розв'язку;
- покладені в основу розробки наближених обчислювальних схем ідеї (метаевристики) дозволяють створювати алгоритми, які можуть розв'язувати не одну, а цілий клас близьких за постановкою оптимізаційних задач;
- важливий клас оптимізаційних задач породжується проблемами з директивним терміном, тобто їх розв'язок має бути знайдений до зазначеного апріорі строку;
- у деяких задачах значення цільової функції можуть бути доступними лише в процесі розв'язання задачі або змінюватися з часом - цей клас утворюють динамічні, або on-line задачі.

Найуживаніші на практиці наближені алгоритми можна поділити на сім класів (рис. 1.2).

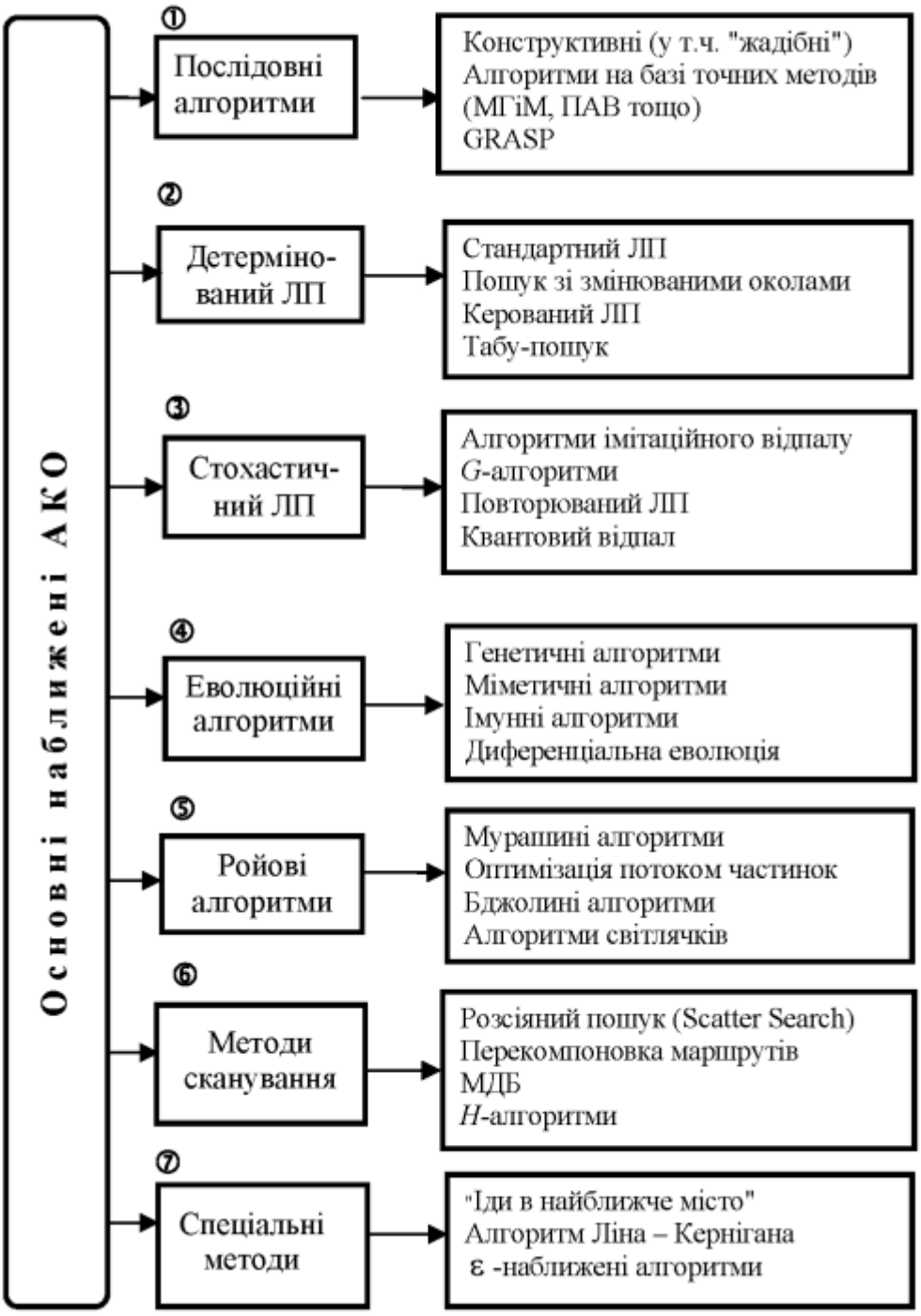


Рис. 1.2. Класифікація основних наближених методів комбінаторної оптимізації

Тут МГіМ - це метод гілок і меж, ПАВ - послідовний аналіз варіантів, ці точні алгоритми використовуються в даному контексті для породження наближених обчислювальних схем. Також варто зазначити, що в різних джерелах поділ і назви змінюються. Так наведені на рисунку ройові алгоритми

в багатьох джерелах звуть популяційними, а значна кількість науковців не розділяє популяційні та еволюційні на різні категорії і через їх схожість зводить до однієї категорії.

Очевидно, що загальна кількість усіх розроблених на сьогодні АКО значно перевищує список, наведений на цьому малюнку. Проте, як показує аналіз наукових публікацій, ці алгоритми та їх модифікації є основними інструментами для вирішення практичних задач дискретної оптимізації. Більшість алгоритмів класів 2-7 є ітераційними методами, баільшість з яких засновані на застосуванні процедур місцевого пошуку. Крім того, метаевристика широко використовується для вирішення проблем з більшою точністю - саме таким алгоритмам і буде приділено основну увагу надалі.[3]

Розглянемо найпоширеніші нечіткі алгоритми.

Алгоритм мурашиної колонії.

Ідея імітації поведінки мурах для пошуку гарних рішень для комбінаторних задач оптимізації була ініційована Доріго, Маньєццо та Фарні. Принцип цих методів ґрунтується на тому, як мурашки шукають їжу і знаходять свій шлях назад до гнізда. Спочатку мурахи досліджують територію навколо свого гнізда випадково. Як тільки мураха знаходить джерело їжі, вона оцінює кількість та якість їжі та поставляє її частину у гніздо. Під час зворотного відправлення мураха залишає хімічний слід із феромонів на землі. Роль цього феромонного сліду полягає в тому, щоб спрямувати інших мурашок у бік джерела їжі, а кількість феромонів, залишених мурахою, залежить від кількості знайдених продуктів. Через деякий час шлях до джерела їжі буде позначений сильним слідом феромонів, і чим більше мурах, які досягають джерела їжі, тим сильніше залишається слід із феромонів.

Оскільки джерела, близькі до гнізда, відвідуються частіше, ніж ті, що далеко, феромонні стежки, що ведуть до найближчих джерел, ростуть швидше. Ці феромонні стежки експлуатуються мурахами як засіб, щоб знайти свій шлях від гнізда до джерела їжі та назад.

Транспозиція реальної поведінки щодо пошуку продуктів харчування в алгоритмічну структуру для вирішення задач комбінаторної оптимізації здійснюється шляхом аналогії між:

– реальною областю пошуку мурах та набором можливих рішень комбінаторної задачі;

– кількість їжі в джерелі та цільовою функцією;

– феромонова трасою та адаптивною пам'яттю.

Генетичні алгоритми.

Одним із різновидів евристичних алгоритмів є генетичні алгоритми. Генетичний алгоритм (ГА) — це еволюційний метод пошуку для вирішення задач оптимізації з використанням механізмів, які імітують природну еволюцію. У GA генерується життєвий цикл організму: відтворення потомства, успадкування та механізми природного відбору для збереження особин на оптимальних стандартах. В *GA* всі терміни пов’язані з біологією і мають спрощений вигляд . *GA* оперує популяціями особин. Особина — об’єкт предметної області вирішуваної проблеми. Кожна особина складається із хромосом — закодованим набором кінцевої множини чисел параметрів задачі, точки в просторі пошуку. Хромосоми в свою чергу складаються із впорядкованої послідовності генів — атомарних базових елементів *GA*.

Мірою пристосованості кожної особини в популяції є функція пристосованості або функція оцінки (*fitness function*). Вона дозволяє вибрати найбільш пристосовані особини (з максимальним значення фітнес-функції) відповідно до еволюційних принципів виживання «найсильніших» (найкраще пристосувалися). Функція відповідності впливає на функцію генетичного алгоритму і повинна мати точне та правильне визначення. У задачах оптимізації функцію пристосування зазвичай максимізують, що називається цільовою функцією. На кожній ітерації генетичного алгоритму функція придатності використовується для оцінки придатності кожної особини в даній популяції, і на основі цього створюється наступна популяція індивідуумів, яка становить набір потенційних рішень проблеми, наприклад , оптимізувати

завдання. Наступна множина елементів в генетичному алгоритмі називається поколінням.

Сам алгоритм складається з декількох кроків.

Підготовчий етап – формування вихідної сукупності (вихідної множини рішень). Отримані алгоритми можуть відрізнятися, але найчастіше використовують стохастичні процеси, щоб охопити більшу різноманітність у пошуку рішень. Можна використовувати й інші методи формування, наприклад, методи з раніше відомими характеристиками, але звернути увагу, що цематеме значний вплив на процес розробки майбутньої системи. Відбір є важливим кроком в алгоритмі, що контролює вибір напрямку розвитку популяції. Рішення з низькими значеннями функції пристосованості найчастіше відхиляються, що покращує загальну середню пристосованість всієї популяції.

Гібридизація — це стадія, на якій у популяції, відібраній для відновлення популяції, утворюються нові розчини. Його характеристика полягає в тому, що при кросинговері беруться два або більше існуючих розчинів у популяції та їх компоненти (гени) об’єднуються в новий розчин, який залишається в популяції.

Гібридизація не дозволяє повністю охопити всі можливі комбінації та значення генів, тому процес мутації є не менш важливим. Вона полягає в тому, що в деяких розчинах популяції стаються різні зміни в генах за випадковим принципом. Цей процес позитивно впливає на збільшення різноманітності особин у популяції.

Оцінка рішення та зупинка алгоритму - у більшості випадків; якщо розвязання проблеми потрібно використовувати генетичний алгоритм, немає критерію зупинки на основі самого рішення, а метод підрахунку часу (кількості створених популяцій) використовується. Зупинка алгоритму іноді може бути згенерована раніше, якщо можлива ситуація з виродженою популяцією.

Табу-пошук.

У 1991 році Тайлард запропонував надійний метод табу. Метод реактивного табу був запропонований в 1994 році . У 2005р. Дрезнер представив новий пошук табу.

Основні ідеї табу-пошуку (*TS*) можна охарактеризувати наступним чином. Перший інгредієнт, який є загальним для більшості евристичних та алгоритмічних процедур, полягає у ви значенні сусідства або набору ходів, які можуть бути застосовані до даного рішення щоб створити новий.

Серед усіх сусідніх рішень *TS* прагне отримати найкращу евристичну оцінку. У найпростішому випадку така оцінка вказує на вибір переміщень, що покращує більшу частину цільової функції. Якщо немає покращених ходів (що вказують на місцевий оптимум), *TS* вибирає той, який найменш погіршує цільову функцію.

Щоб уникнути повернення до місцевого оптимуму, який тільки що був відвіданим, перехресний перехід повинен бути заборонений. Це відбувається шляхом збереження цього переміщення (або, точніше, характеристики даного переміщення) у структурі даних, яка називається табу-списком. Цей список містить кількість елементів, що визначають заборонені (табу) ходи. Параметр s називається розміром списку табу. Вибір цього розміру табу-списку є критичним. Багато досліджень зроблено для визначення оптимального чи хорошого розміру цього списку табу, а автори пропонують різні підходи, включаючи фіксований розмір та динамічний розмір.[6]

Алгоритм моделювання відпалу — це загальний метод алгоритму для вирішення глобальних задач оптимізації, особливо дискретних і комбінаторних задач оптимізації. У фізиці конденсування середовища відпалу — це термічний процес отримання низькоенергетичного стану об’єкта в термічній ванні (кріостаті).

Цей процес складається з двох етапів:

- збільшити температуру термічної ванни до максимуму для плавлення в твердому тілі;

- повільно знижуйте температуру ванни з гарячою водою, доки частинки не вирівняються до основного стану тіла.

У рідкій фазі частинки розташовані безладно, а в твердому основному стані всі частинки розташовані в добре структурованій ґратці з відповідною мінімальною енергією. Основний стан твердого тіла можна отримати, лише якщо максимальне значення температури є достатньо високим, а охолодження відбувається досить повільно. Інакше тіло замерзне в метастабільному стані замість реального.

Алгоритм заснований на моделюванні фізичних процесів, що відбуваються під час переходу речовини з форми рідини в твердий матеріал, у тому числі під час відпалу металу. Вважається, що атоми вже організовані в решітку, але перехід окремих атомів з однієї елементарної комірки в іншу все ще допускається. Передбачається, що цей процес протікає з поступовим зниженням температури. Перехід атомів з одного положення в інше відбувається з певною ймовірністю, і ймовірність зменшується зі зниженням температури. Стабільна решітка відповідає мінімальній енергії для атомів, тому атоми або переходять у нижчий енергетичний стан, Або залишайтеся на місці. (Цей алгоритм також відомий як алгоритм Метрополіса, названий на честь його автора Ніколаса Метрополіса). Моделюючи такий процес, можна знайти точку або набір точок, у яких досягається найнижче значення певної числової функції. Повертаючись до імітації відпалу, алгоритм Метрополіса може бути використаний для генеруваня послідовності рішень задачі комбінаторної оптимізації, припускаючи еквівалентності між фізичною системою багатьох частинок і завданням комбінаторної оптимізації:

- розв'язання задачі комбінаторної оптимізації еквівалентні стану фізичної системи;
- вартість рішення еквівалентно енергії стану системи.

Крім того, ми вводимо керуючий параметр, який відіграє роль температури. Таким чином алгоритм імітації відпалу, можна розглядати як ітерації алгоритму Метрополіса, виконані на зменшення значення керуючого

параметра. Значення чотирьох функцій в процедурі *SIMULATED_ANNEALING* очевидні:

- *INITIALIZE* обчислює і задає початкові значення параметрам с та L;
- *GENERATE* вибирає сусідій розв'язок відносно поточного розв'язку;
- *CALCULATE.LENGTH* і *CALCULATE_CONTROL* обчислює нові значення параметрів *L* та с відповідно.

В порувнянні *simulated annealing* та ітераційного методу, зрозуміло що *simulated annealing* можна вважати за узагальнення. Коли значення контрольного параметра дорівнює нулю, алгоритм моделювання відпалу стає таким самим, як ітераційне вдосконалення. Що стосується продуктивності обох алгоритмів, імітований відпал виконується швидше, ніж ітераційне вдосконалення для більшості завдань.

Остаточний час моделювання відпалу отримується шляхом створення серії однорідних ланцюгів Маркова кінцевої довжини зі зменшенням контрольних параметрів. Для цього набору параметрів слід задати фактори, що визначають збіжність алгоритму. Ці параметри поєднуються в спеціальну функцію, яку іменують графіком охолодження. Графік зниження температури на пряму впливає на кінцеву послідовність значень параметрів і кінцеву кількість переходів для кожного значення контрольного параметра. Точніше, він визначає:початкове значення параметра контролю c_0;

- зменшення функції для зниження вартості контрольного параметра;
- кінцеве значення контрольного параметра, вказаний критерій зупинки,також скінченне значення довжини кожного однорідного ланцюжка Маркова.[7]

ВИСНОВКИ ДО РОЗДІЛУ 1

В першому розділі детально розглянуто методи вирішення задачі дискретної оптимізації. Також були проаналізовані найпопулярніші на даний момент алгоритми для дискретної оптимізації, способи та особливості їх побудови та використання.

В результаті дослідження були зроблені певні висновки:

1. Точні алгоритми найпростіші в плані побудови і реалізації, але мають дуже низьку ефективність. Це свідчить про те, що їх доцільно використовувати лише як частину іншого алгоритму оптимізації, або в ситуації коли потрібна максимальна точність результату незалежно від часу затраченого на його отримання чи область рішень досить мала і використання складніших алгоритмів не має сенсу.

2. Неточні алгоритми більш ефективні ніж точні стосовно витраченого часу, але набагато складніші в реалізації. Крім цього, ми не можемо бути впевнені що результат буде абсолютно точний, але цей недолік нівелюється можливістю тонкого настроювання отримуваних результатів, наприклад, в задачі комівояжера отримати не лише найкоротший або найдешевший(при внесенні ціни проїзду) шлях, але і найкоротший шлях, що не дорожчий заданої суми.

Важливо вказати що задача дискретної оптимізації має велику цінність у багатьох сферах життя, від побудови маршрутів громадського транспорту до створення шкільного розкладу.

РОЗДІЛ 2. ОГЛЯД АЛГОРИТМІВ ТА ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Огляд алгоритмів для оптимізації

Далі обговорюються алгоритми, що використовуються для оптимізації, такі як генетичний алгоритм, алгоритм світлячка, алгоритм оптимізації бур'янів, пошук зозулі, алгоритм, натхненний кажаном, алгоритм пошуку мавпи та алгоритм риб'ячого рою.

2.1.1 Генетичний алгоритм

Генетичний алгоритм - це метод багатовимірної оптимізації, тобто метод пошуку мінімуму багатовимірної функції. Потенційно цей підхід можна використовувати для глобальної оптимізації, але це має певні ускладнення.

Суть методу полягає в тому, що ми регулюємо еволюційний процес: маємо деяку племінну популяцію (набір векторів), що піддається мутаціям, а природний відбір заснований на мінімізації цільової функції. Розглянемо ці процеси докладніше.

Отже, по-перше, наше населення має збільшуватися. Основний принцип розмноження полягає в тому, що нащадки схожі на своїх батьків. Тобто ми повинні вказати якийсь механізм успадкування. Було б краще, якби він містив елемент випадковості. Але швидкість розвитку таких систем дуже низька -- Генетичне різноманіття зменшується, а популяції вироджуються. Тобто значення функції більше не мінімізується.

Для вирішення цієї проблеми вводиться мутаційний механізм, тобто випадкова мінливість деяких особин. Цей механізм дозволяє вводити нове в генетичне різноманіття.

Наступний важливий механізм – відбір. Як було сказано раніше, відбір - це відбір індивідів (вибирати можна тільки з новонароджених, але можна і з усіх людей - практика показала, що це не грає вирішальної ролі), і функцію краще звести до мінімуму.

Зазвичай відбирають стільки особин, скільки перед розведенням, тому кількість особин у популяції є постійною від однієї ери до іншої. Також використовується для відбору «щасливчиків» — певної кількості людей, які можуть сильно зменшити функціональність, але принести різноманітність наступному поколінню.

Цих трьох механізмів зазвичай недостатньо для мінімізації функціональності. Ось як популяція вироджується - рано чи пізно місцевий мінімум вбиває все населення за те, чого він вартий. Коли це відбувається, відбувається процес, званий землетрусом (по суті аналогії - глобальна катастрофа), під час якого майже вся популяція знищується, і додаються нові (випадкові) особини. [8].

Розглянемо основні відмінності генетичних алгоритмів від традиційних.

1. Генетичний алгоритм використовує код, який забезпечує набір параметрів, які залежать від параметрів цільової функції. Крім того, інтерпретація цих кодів відбувається лише перед запуском алгоритму та після його завершення. Під час виконання операції з кодами повністю не залежать від їх інтерпретації, код розглядається лише як бітовий рядок.

2. Генетичні алгоритми використовують кілька точок простору пошуку одночасно, але не точково, як деякі традиційні методи. Це дає змогу подолати один із їхніх недоліків – якщо функція не є унімодальною, тобто має множинні екстремуми, існує ймовірність потрапляння в локальні екстремуми цільової функції..

3. Генетичний алгоритм не використовує додаткову інформацію в процесі роботи, що підвищує швидкість роботи. Єдина інформація, яка використовується, це діапазон дозволених значень параметрів і цільова функція в будь-якій точці.

4. Генетичні алгоритми використовують як ймовірнісні правила для появи нових точок аналізу, так і детерміновані правила для переходу від однієї точки до іншої. Паралельне використання випадкових і детермінованих елементів дає набагато більший ефект, ніж кожен окремо.

На початку генетичного алгоритму формується перше покоління особин. Далі вибираються кілька елітних осіб (якщо включені відповідні політики) шляхом обчислення та порівняння значень цільової функції для кожної особи. Елітні особи — це ті, хто досягає найменшої цінності (наприклад, довжина шляху комівояжера).

Існує три види кросоверів: два варіанти одноточкового кросовера і двоточковий кросовер. Підбір схрещених пар виконується за допомогою генератора випадкових чисел. На наступному етапі нове покоління особин з певною ймовірністю мутує. Під час скорочення розширена популяція зменшується до початкового розміру шляхом видалення з неї особин з найгіршими значеннями об’єктивної функції. Процес еволюції закінчується, якщо оптимальне значення цільової функції не змінюється протягом заданої кількості поколінь.

2.1.2 Алгоритм світлячків

У світі існує близько 2000 видів світлячків, і більшість із них мають здатність випромінювати короткі ритмічні спалахи світла. Спалахи утворюються в процесі біолюмінесценції. Основною функцією таких спалахів вважається залучення партнерів і потенційних жертв. Крім того, сигнальні спалахи можуть служити захисним механізмом, попереджаючи потенційних хижаків про те, що світлячки гіркі на смак. Деякі тропічні світлячки можуть синхронно блимати, демонструючи приклад біологічної самоорганізації. Інтенсивність світла як функція відстані від джерела світла підкоряється закону обернених квадратів. Крім того, зі збільшенням відстані r інтенсивність світла зменшується внаслідок поглинання світла повітрям. Поєднання цих двох факторів визначає, на якій відстані світлячки можуть бачити один одного,

а це кілька сотень метрів вночі. Відомі два варіанти алгоритму оптимізації популяції, натхненного поведінкою світлячків - алгоритм *Firefly* і алгоритм оптимізації *Firefly swarm*. Ключова відмінність світлячків від світлячків полягає в тому, що останні не мають крил.

Алгоритм світлячка (F-алгоритм) був запропонований Янгом у Кембриджському університеті в 2007 році. Алгоритм використовує таку модель поведінки світлячка [10]:

- всі світлячки можуть притягувати один одного незалежно від статі;
- притягання світлячків до інших особин пропорційно їх яскравості;
- менш привабливі світлячки рухаються в напрямку більш привабливих світлячків;
- сяйво цього світлячка, видиме іншим світлячком, зменшується зі збільшенням відстані між світлячками;
- світлячок рухається навмання, якщо не бачить біля себе більш яскравого світлячка.

Алгоритм *Firefly* є одним із широко використовуваних евристичних алгоритмів. Його прості та легкі кроки привабили дослідників з різних дисциплін своєю ефективністю.

2.1.3 Алгоритм бур’янистої оптимізації

Алгоритм бур’янистої оптимізації (*Invasive Weed Optimization*) був створений на основі такого звичайного явища, як бур’яни, що вторгаються на сільськогосподарські поля. Алгоритм, запропонований іранськими вченими Меграбіаном (AR Mehrablan) і Лукасом (S. Lucas) у 2006 році, базується на моделюванні ознак, таких як посів, ріст і конкуренція в популяціях бур’янів.

Простіше кажучи, бур'ян - це будь-яка рослина, яка росте там, де вона не потрібна. Взагалі бур'яном можна вважати будь-яку рослину. Однак цей термін зазвичай використовується для позначення тих рослин, експансивність яких становить серйозну загрозу для культурних рослин. У зв’язку з високою

актуальністю боротьби з бур’янами у світі випускається велика кількість преси, повністю присвяченої систематиції бур’янів, їх екології та фізіології, методам боротьби тощо. Найцікавішою особливістю проблеми бур’янів є широко поширене переконання, що вони завжди перемагають, і чим більше людей намагаються їм протистояти, тим більше вони розмножуються [11].

Просторові можливості для поширення бур’янів створюють людські сільськогосподарські системи. Бур'яни проникають у ці простори, спочатку розсіюючи своє насіння, потім колонізуючи і, нарешті, займаючи поле. Біологічне різноманіття бур'янів і їх висока пристосованість до місцевих умов забезпечують високу ефективність цих процесів. Залежно від виду бур'яну він може розмножуватися з гаметами або без них.

Статеве розмноження відбувається за допомогою насіння, яке утворюється в материнській рослині після запилення яєць. Потім ці насіння поширюються вітром, водою, тваринами тощо (поширення у просторі). Якщо насіння потрапить у придатне для життя середовище, воно проросте, виросте у дорослу рослину, а потім зацвіте та зав’яже насіння.

Основним механізмом, що визначає динаміку будь-якого рослинного угруповання, є природний відбір, з якого виділяють два крайніх типи: r-відбір і K-відбір. Справжня стратегія вибору лежить між цими типами країв. Можна сказати, що девіз r-відбору живи швидко, розмножуйся швидко, помри молодим. Цей тип відбору необхідний для досягнення успіху в мінливому, непередбачуваному середовищі. r-селекція має чудові характеристики, такі як висока плодючість, дрібне насіння та здатність до передачі на великі відстані. K-відбір приймає принцип live slow, reproduce slow, die old (живи повільно, розмножуйся повільно, помри старим).

Цей тип відбору необхідний для успіху в стабільному, передбачуваному середовищі, де конкуренція за обмежені ресурси між конкуруючими особами може бути інтенсивною. Це може статися, якщо чисельність популяції в середовищі існування близька до максимальної, яку вона може прийняти.

Розглянемо задачу загальної максимізації фітнес-функції. В алгоритмі бур'янистої оптимізації спосіб поведінки бур'янів захопленні враховує такі базові властивості процесу.

1. Розподіл загальної кількості насіння по всій області пошуку (ініціалізація популяції).
2. Створення насіння в залежності від пристосованості рослин (відтворення).
3. Розподіл виробленого насіння у випадковому порядку по області пошуку (просторовий розподіл).
4. Повторення кроків 2, 3 до тих пір, поки не буде досягнутий заданий максимум числа рослин.
5. Відбір рослин з більш високою пристосованістю, їх відтворення і просторовий розподіл (конкурентний виняток).
6. Повторення кроку 5 до виконання умови закінчення процесу.

Тому рослини та їхнє потомство оцінюють разом, причому до розмноження допускають ті, які краще підходять. Цей механізм дозволяє рослинам з низькою пристосованістю розмножуватися, а їх нащадкам виживати, якщо вони добре пристосовані. [12].

2.1.4 Зозулин пошук

Зозулин пошук (*Cuckoo Search*) – алгоритм що був натхненний поведінкою зозуль, які відкладають свої яйця в гнізда інших птахів.

В алгоритмі CS кожне яйце в гнізді є розвязком, а яйце зозулі – новим розвязком. Мета полягає в тому, щоб замінити не дуже хороший розчин у гнізді новим і, можливо, кращим (зозулиним) розчином. У найпростішому варіанті цього алгоритму в кожному гнізді знаходиться одне яйце.

Припустимо, мова йде про задачу глобальної безумовної оптимізації, де функція пристосованості повинна бути максимізована. Алгоритм базується на наступних трьох правилах [13]:

- усі зозулі відкладають по одному яйцю за раз у випадково вибране гніздо;
- кращі гнізда з яйцями високої якості (високим значенням придатності) переходять в наступне покоління;
- яйце зозулі, що було відкладене в гніздо, може бути знайдено господарем з певним встановленим шансом $\xi_a \in (0;1)$ і видалено з гнізда.

Схема алгоритму CS:

1. Ініціалізуємо популяцію $S=s_i, i \in [1:|S|]$ з $|S|$ хазяйських гнізд і зозулю, тобто визначаємо вектори $X0i$ і вектор початкового положення зозулі Xc.

2. Виконуючи польоти Леві, знаходимо нове положення зозулі за формулою:

$$X_c' = X_c + V \otimes L_{|X|}(\lambda),$$

де $V=(vj, j \in [1:|X|])$ – вектор розміру кроків по відповідним компонентам вектора X;

$L|X|(\lambda)-(|X|\times 1)$ – випадковий вектор незалежних дійсних випадкових чисел, розподілених за законом Леві.

3. Випадковим чином вибираємо гніздо sj, $j \in [1:|S|]$ і, якщо $\varphi(Xc) > \varphi(Xj)$, замінюємо яйце в цьому гнізді на яйце зозулі, тобто вважаємо $Xj=Xc$.

4. З імовірністю ξa ми видаляємо певну кількість випадково вибраних бідніших гнізд (можливо, включаючи гнізда sj) із популяції та будуємо таку саму кількість нових гнізд відповідно до правил кроку 1.

Випадково рівномірно розподіляємо початкові положення гнізд і зозуль у деякому гіперпаралелепіпеді. Загалом, усі компоненти вектора V вважаються ідентичними та дорівнюють v, де значення v пов'язане з протяжністю області пошуку. Основними вільними параметрами алгоритму є константи, які визначають траєкторію польоту Леві та ймовірність ξa видалення гнізда.

2.1.5 Алгоритм, інспірований кажанами

Алгоритм, інспірований кажанами (*Bat-Inspired, BI*), запропонований Янгом в 2010 році. Алгоритм може здатися більш складним, за переважну множину інших алгоритмів ройового інтелекту, а також еволюційних алгоритмів, але його можна дуже ефективно застосовувати до задач оптимізації та дає хороші результати, займаючи менше часу [14].

Більшість видів кажанів мають складні методи ехолокації, якими вони користуються для виявлення їжі та перешкод, а також для легкого нересування в темряві. Для локалізації рухомих об'єктів миші використовують ефект Доплера [15]. Алгоритм BI передбачає наступну модель поведінки кажанів.

За допомогою ехолокації всі миші можуть вимірювати відстань до здобичі і перешкод, а також розрізняти їх.

1. Мишка рухається випадково. Поточна позиція та швидкість миші Si дорівнюють Xi і V відповідно. Щоб знайти здобич, миші можуть змінювати частоту сигналу, а також частоту повторення (частоту пульсу) випромінюваних імпульсів.

2. Частота сигналу може змінюватися в діапазоні [ωmin,ωmax], $\omega max > \omega min \geq 0$, гучність сигналу - в діапазоні від 0 до 1.

Припустимо, мова йде про задачу глобальної безумовної мінімізації функції. Нижче наведено основні кроки схеми алгоритму BI.

1. Ініціалізація заповнення. Встановлюємо початкову позицію агента Si.

2. Ми ідентифікуємо глобально оптимального агента та його відповідний розв’язок $X**$.

3. Проксі-міграція. Ми переміщуємо всі проксі-сервери на один крок відповідно до використовуваної процедури міграції.

4. Локальний пошук. Використовуючи ймовірність ξir (вільний параметр алгоритму), ми реалізуємо процедуру локального пошуку в околиці найкращого рішення $Xi*$, знайденого агентом Si для всіх попередніх ітерацій. Ми приймаємо знайдене рішення як нову поточну посаду агента Si.

5. Глобальний пошук. В околицях поточного рішення X ми випадково генеруємо розв'язки Xi'. Якщо $\varphi(Xi')<\varphi(X)$, то ми приймаємо рішення Xi' як нову поточну посаду агента Si з імовірністю ξia. Ми знаходимо нове глобально краще рішення X.

6. Еволюція параметрів ξir і ξia.

7. Завершіть ітерацію. Перевіряємо, чи виконується умова завершення ітерації.

Якщо ці умови задовольняються, ми приймаємо поточну точку $X**$ як розв'язок і завершуємо обчислення, інакше повертаємось до кроку 2.

Ми ініціалізуємо популяцію шляхом випадкового розподілу агентів Si у цій частині області пошуку. На даному етапі ми встановлюємо початкові значення частоти ωi, гучності ai і частоти повторення імпульсів ri, рівномірно випадково розподілених у відповідному інтервалі $[\omega min,\omega max]$, $[amin,amax]$, [0; 1].

Міграція агентів. Міграцію агента si здійснюємо за формулами:

$$X_i'=X_i+V_i',$$

$$V_i'=V_i+w_i' (X_i-X^{**}),$$

$$\omega_i=\omega^{min}+(\omega^{max}-\omega^{min})\cup_1(0;1)$$

Іншими словами, відповідно до процесу міграції напрямок руху агента визначається сумою вектора зміщення (член Vi) попередньої ітерації та вектора напрямку випадкового збурення для досягнення оптимального агента ($Xi-X**$).

Локальний пошук виконуємо за наступною схемою.

Випадковим чином варіюємо поточний стан агента Si відповідно до формули.

$$X_i'=X_i+\bar{a}\cup_{|x|}(-1,1),\ i\in[1:|S|],$$

де $\bar{a}$ – поточне середнє значення гучності всіх агентів популяції:

$$\bar{a}=\frac{1}{|S|}\sum_{i=1}^{|S|}a_i$$

Обчислюємо значення функції в новій точці. Якщо більше, завершіть процес локального пошуку, інакше поверніться до кроку 1.

Автори алгоритму провели масштабні експериментальні порівняння ефективності алгоритму BI з генетичними алгоритмами та оптимізацією роїв частинок. Результати показують, що з точки зору глобальної екстремальної ймовірності локалізації, алгоритм BI є більш ефективним, ніж вищезгадані алгоритми.

2.1.6 Алгоритм мавпячого пошуку

Алгоритм був натхненний поведінкою мавп, які здатні лазити по деревах у пошуках їжі. Мавпи зіставляються з агентами, які досліджують поверхню функції відповідності, щоб знайти екстремуми в глобальній проблемі максимізації. Вважається, що правила поведінки мавп такі: чим вище гора, тим більше їжі на вершині.

Кожна мавпа рухається вгору зі свого поточного положення, поки не досягне вершини пагорба. Потім мавпа виконує серію часткових стрибків у будь-якому напрямку, сподіваючись знайти вищу гору, і повторює рух вгору.

Здійснивши фіксовану кількість підйомів і локальних стрибків, мавпа вважала, що повністю дослідила ландшафт поблизу свого початкового місця. Щоб дослідити нові регіони пошукового простору, мавпа здійснила довгий глобальний стрибок.

Зазначена вище дія повторюється задану кількість разів. Рішення проблеми оголошується як найвища вершина, знайдена для даної популяції мавп.

Схему М-алгоритму можна представити в наступному вигляді:
Ініціалізацію популяції мавп $si,i\in[1:|S|]$ виконують за загальними правилами, а саме шляхом рівномірного випадкового розподілу агентів у пошуковому просторі.

Процес руху вгору (*climb process*) це процес локального пошуку, який можна реалізувати кількома способами. В оригінальному алгоритмі автори

пропонують використовувати алгоритм, заснований на процедурі стохастичної апроксимації [19].

Локальні стрибки (*watch-jump process*). Схема процесу локальних стрибків для агента $si, i \in [1:|S|]$ має наступний вигляд:

1. Виходячи з поточного стану агента Xi, генеруємо його нове можливе положення X_i' за формулою:

$$x'_{i,j} = U_1\left((x_{i,j} - b);(x_{i,j} + b)\right), i \in [1:|S|], j \in [1:|X|],$$

тобто вважаємо, що по кожній з координат нове положення агента являє собою випадкову величину, рівномірно розподілену в інтервалі $[(x_{i,j}-b);(x_{i,j}+b)]$. Тут $b > 0$ - максимально можлива довжина стрибка (вільний параметр алгоритму).

2. Якщо точка Xi' є допустимою (належить області допустимих значень D) і $\varphi(Xi') \geq \varphi(X)$, то вважаємо $Xi = Xi'$ і завершуємо для даного агента локальні стрибки, в іншому випадку задане число раз повертаємося до кроку 1.

Глобальні стрибки (somersault process) агенти здійснюють зі свого поточного положення у напрямку поточного центру ваги їх координат за наступною схемою.

1. Генеруємо випадкове дійсне число $v = U1(v-; v+)$, що має величину кроку глобального стрибка. Тут $v-, v+$ - нижня і верхня межі цієї величини, які мають в загальному випадку різні знаки, так що величина v може бути як позитивною, так і негативною.

2. Знаходимо нове можливе положення Xi' агента si' за формулою:

$$x'_{i,j} = x_{i,j} + v\left(x_j^c - x_{i,j}\right), i \in [1:|S|], j \in [1:|X|],$$

де $x_j^c = 1|S|\Sigma xi,j|S|i=1$ - поточний стан центра тяжіння агентів популяції по j-му координатному напрямку.

3. Якщо положення Xi' є допустимим, то вважаємо $Xi = Xi'$ і завершуємо для даного агента глобальні стрибки, інакше задане число раз повертаємося до кроку 1.

Зауважте, що якщо розмір кроку v глобального стрибка приймає позитивне значення, агент стрибає до центру ваги Xc, якщо значення від'ємне – у протилежному напрямку.

Як критерій завершеності ітерації ми використовуємо реалізації заданої кількості повторень цих алгоритмічних дій, як описано вище.

Оскільки алгоритм M не гарантує, що оптимальне рішення буде знайдено на останній ітерації, поточне оптимальне рішення потрібно зберігати в пам'яті комп'ютера щоразу. [17].

2.1.7 Алгоритм косяка риб

Алгоритм Fish School Search (*FSS*) був запропонований Філо і Нето в 2008 році. В алгоритмі *FSS* риби плавають в акваріумі (зоні пошуку) у пошуках їжі. Вага кожної риби формалізує її особистий успіх у пошуку рішень і функціонує як пам'ять. Головною особливістю парадигми *FSS* є саме наявність вагових коефіцієнтів у сукупності суб'єктів. Оператори алгоритму *FSS* об'єднані в дві групи [17]:

- оператор годування, який формалізує успішність дослідження рибами тих чи інших областей акваріума;
- оператори плавання, які реалізують алгоритми міграції агентів.

Оператор подачі. Нехай Wi буде поточною масою предмета Si. В алгоритмі FSS передбачається, що вага агента пропорційна нормалізованій різниці значення функції пристосованості між наступною ітерацією та поточною ітерацією.

Оператори плавання (swimming operators).

В алгоритмі FSS виділяють три типи плавання – індивідуальне, інстинктивно-колективне та колективно-вольове. Ці види плавань виконуються на інтервалах $(t,\tau]$,$(\tau,\theta]$,$(\theta,t']$ основного ітераційного інтервалу $(t,t']$ відповідно.

Індивідуальне плавання (individual swimming).

У цьому випадку напрямок руху агента також може бути випадковим. Якщо це переміщення виводить агента за межі діапазону дозволених значень для D, ми не виконуємо переміщення. Так само, якщо значення функції пристосування нової точки $Xi\tau$ не перевищує значення функції пристосованості попередньої точки Xit, агент Si не переміститься. Процес індивідуального плавання може складатися не з однієї ітерації, а з певної кількості ітерацій. Таким чином, особисту навігацію агента можна інтерпретувати як локальний пошук поблизу поточного місцезнаходження агента.

Інстинктивно-колективне плавання (*collective-instinct swimming*) буде виконано за формулою після того, як усі агенти завершать індивідуальне плавання:

$$W_i' = W_i + \frac{\varphi(X_i') - \varphi(X_i)}{max(\varphi(X_i'), \varphi(X_i))}, \qquad i \in [1:|S|]$$

Другий член у формулі — це не що інше, як крок міграції, спільний для всіх агентів, який є зваженою сумою індивідуальних рухів агентів. Наведена вище формула означає, що під час інстинктивного колективного плавання на кожного агента впливають усі інші агенти в популяції, і цей вплив пропорційний індивідуальному успіху агента.

Колективно-вольове плавання (*collective volition swimming*) виконуємо слідом за інстинктивно-колективним плаванням. Якщо загальна вага косяка збільшується за рахунок індивідуального та інстинктивного колективного плавання, колективне добровільне плавання полягає в переміщенні всіх агентів у напрямку до центру жорсткості поточної популяції, і в протилежному напрямку - якщо ця вага зменшується. Іншими словами, при середньому успішному запливі популяція притягується до свого центру ваги, тобто інтенсивність пошуку зростає. В іншому випадку популяція розширюється з того самого центру, збільшуючи свої різноманітні властивості.

Відомо багато вдосконалених алгоритмів *FFS*. Для прикладу можна розглянути так званий щільний алгоритм *FFS (Density FFS, DFFS)*, який

використовує модифікації операторів класичного алгоритму, а також нові оператори – так звані оператори пам’яті та розділення. Оператор пам’яті будується на векторі - Mi, який заповнює агентів і формалізує поточний набір попередніх впливів інших агентів популяції на кожного даного агента. Оператор поділу використовує пам'ять агента та прагне сформувати кілька субпопуляцій відповідно до популяції [19].

2.2 Порівняльний аналіз еволюційних алгоритмів

Далі будуть описані у табл. 2.1. переваги та недоліки генетичного алгоритму, алгоритму світлячків, алгоритму оптимізації бур’янів, пошуку зозулі, алгоритму пошуку мавпи, алгоритму рибного зграї та інших глобальних алгоритмів оптимізації.

Таблиця 2.1

Алгоритм	Переваги	Недоліки
Генетичний алгоритм	3. Легко модифікується. 4. Підтримка багатоцільової оптимізації. 5. Легко паралелізуєтьс. 6. Використовують правила імовірнісного переходу. 7. Надійний до локальних мінімумів / максимумів.	1. Обчислювально дорогий. 2. Будь-який невідповідний параметр ускладнить сходження алгоритму або просто дасть безглузді результати.
Алгоритм світлячків	1. Здатність мати справу з мультимодальністю. 2. Автоматичний підрозділ (популяції автоматично підрозділяються на підгрупи).	1. Висока обчислювальна часова складність. 2. Повільна швидкість збіжності.

Алгоритм	Переваги	Недоліки
	3. Параметри можуть бути налаштовані для управління випадковістю під час ітерацій.	
Алгоритм бур'янистої оптимізації	1. Всі можливі кандидати беруть участь у процесі відтворення . 2. Алгоритм є простим і включає меншу кількість обчислювального навантаження.	1. При збільшенню пошукового простору збільшується час обчислення та кількість змінних, що підлягають налаштуванню. 2. Поступове зменшення стандартної дисперсії може призвести до незрілої конвергенції.
Зозулин пошук	1. Алгоритм простий в застосуванні . 2. Алгоритм залежить від меншої кількості параметрів і є надійним. 3. Алгоритм забезпечує кращу продуктивність для локального пошуку.	1. Дуже легко потрапляє в місцеві оптимальні рішення . 2. Повільна швидкість конвергенції.

Алгоритм	Переваги	Недоліки
Алгоритм, інспірований кажанами	1. Алгоритм простий, гнучкий та простий у реалізації . 2. Може забезпечити дуже швидку конвергенцію на самому початковому етапі. 3. Алгоритм використовує контроль параметрів, який може змінювати значення параметрів у процесі ітерацій . 4. Може ефективно вирішувати масштабні проблеми.	1. Неправильне налаштування параметрів може призвести до застою після певної початкової стадії. 2. Відсутність хороших дослідницьких здібностей.
Алгоритм мавпячого пошуку	1. Алгоритм має кілька параметрів для налаштування, що робить його особливо простим у реалізації. 2. Алгоритм може ефективно уникнути потрапляння в оптимальні локальні рішення . 3. Розмір популяції майже не чутливий до виміру проблем .	1. Оскільки кількість скелелазінь велика, у процесі підйому витрачається багато часу. 2. Можливий невпорядкований напрямок підйому, що веде до невдалого зближення .

Алгоритм	Переваги	Недоліки
Алгоритм косяка риб	1. Прості обчислення у всіх особин. 2. Вирішує завдання глобальної оптимізації для функцій зі складним ландшафтом та великим простором пошуку. 3. Автономність (тобто здатність до самоконтролю функціонування). 4. Масштабованість (з точки зору складності завдань оптимізації / пошуку).	1. Будь-який невідповідний параметр може призвести до отримання невірного рішення.

На основі порівняння можна зрозуміти, що найбільші переваги і найменші мінуси має генетичний алгоритм і алгоритм косяка риб. На мою думку алгоритм косяка риб більше підходить для глобальної оптимізації з пошуком екстремумів функції, а от еволюційний генетичний алгоритм краще підходить саме для дискретної оптимізації.

2.3 Мова програмування Java

Для розробки системи мною було вирішено використати мову програмування Java. Тому я вважаю доречним розкрити переваги, недоліки та особливості даної мови програмування.

У технології мови Java поєднані три ключові елементи, що відрізняє її від усіх існуючих мов програмування, які існують сьогодні.

1. Java надає свої аплети (малі програми) для широкого використання — невеликі, надійні, динамічні програми, які не залежать від платформи

комп’ютера, активні мережеві програми, вбудовані у веб-сторінки. Аплети Java можна створювати та розповсюджувати так само легко, як будь-який документ HTML.

2. Java розкриває можливості об’єктно-орієнтованого створення додатків, що поєднує в собі простий і знайомий синтаксис з надійним та зручним середовищем розробки. Це дозволяє великій кількості програмістів швидко та ефективно створювати нові системи та нові аплети.

3. Java надає програмістам багатий набір класів об'єктів для чіткої абстракції багатьох системних функцій, які використовуються під час роботи з вікнами, мережею, введенням і виведенням. Ключовою особливістю цих класів є те, що вони забезпечують незалежні від платформи абстракції для широкого діапазону системних інтерфейсів.

Наразі мова розробляється компанією Oracle, яка придбала Sun Microsystems у 2009 році. Синтаксис мови в основному походить від C і C++. В офіційній реалізації програми Java компілюються в байт-код, який під час виконання інтерпретується віртуальною машиною конкретної платформи.

Oracle надає компілятор Java і віртуальну машину Java, які відповідають специфікації Java Community Process згідно з GNU General Public License.

Мова запозичує багато свого синтаксису з C і C++. Зокрема, він базується на модифікованій об’єктній моделі C++. Виключає можливість певних конфліктних ситуацій, які можуть виникнути через помилку програміста, і спрощує процес розробки об'єктно-орієнтованих програм. Багато операцій, які програміст повинен виконувати в C/C++, призначаються віртуальній машині. По-перше, Java була розроблена як платформа-незалежна мова, тому вона має менше можливостей низького рівня для роботи з апаратним забезпеченням. Якщо вам потрібно виконати такі операції, java дозволяє викликати підпрограми, написані іншими мовами програмування.

Java вплинула на розвиток J++, розробленого Microsoft. Робота над J++ була зупинена судовим позовом від Sun Microsystems, оскільки мова програмування була модифікованою версією Java. Пізніше на новій платформі

Microsoft .NET було випущено J#, щоб програмістам J++ або Java було легше перейти на нову платформу. З часом нова мова програмування, C#, стала основною мовою платформи, запозичивши багато речей з Java. J# нарешті було включено до Microsoft Visual Studio 2005. Мова сценаріїв JavaScript має подібну назву та синтаксис до Java, але не має нічого спільного з Java.

Одним із основних принципів розробки мови Java є забезпечення захисту від несанкціонованого доступу. Програми Java не можуть викликати глобальні функції та отримувати доступ до довільних системних ресурсів, що забезпечує Java рівень безпеки, недоступний в інших мовах. Цей рівень безпеки для виконання програм Java забезпечує віртуальна машина Java, вбудована в операційну систему. Об'єктна модель Java проста і легко розширюється, але прості типи даних Java не є об'єктами з міркувань продуктивності.

Всесвітня павутина вивела Java на передовий край програмування, а Java, у свою чергу, значно вплинула й навіть змінила вигляд Інтернету, розширивши спектр об’єктів, які можна було поширювати у кіберпросторі. Нова форма програми - аплет - завантажується з віддаленого сервера і може запускатися динамічно, тобто без втручання користувача. До Java цей підхід був неприйнятним з міркувань безпеки та портативності. В архітектуру аплетів вносяться деякі штучні обмеження, що робить їх абсолютно безпечними. По-перше, Java є інтерпретованою мовою, а простір ресурсів програми Java обмежений так званою віртуальною машиною Java (VJM), яка контролює поведінку програми та захищає систему від невеликих програмних помилок, які можуть спричинити побічні ефекти. Крім того, в мові Java існують додаткові обмеження, які не роблять аплети «троянським конем». Зокрема, аплети Java не можуть отримати доступ до локального жорсткого диска. Ця спроба породжує виняток.

Оскільки аплети Java інтерпретуються, а не компілюються, їх виконання на різних платформах набагато легше. У цьому випадку достатньо створити виконувану систему Java для кожної платформи. Якщо така система існує для

даної операційної системи, будь-яка програма Java може працювати в цьому середовищі без додаткової компіляції на цій платформі. Однак Java не є інтерпретованою мовою в чистому сенсі цього слова. Програма Java скомпільована. Результатом роботи компілятора Java є байт-код. Байт-код — це оптимізований набір команд, призначених для виконання віртуальним пристроєм (віртуальною машиною Java). Таким чином вартість інтерпретації зведена до мінімуму, оскільки байт-код уже оптимізовано, і досягається значно висока продуктивність програми Java. Наведені вище характеристики дають нам підстави вважати Java самостійною інформаційною технологією, а не іншою мовою програмування. Тому інтерпретація є найпростішим способом транспортування програм, реалізованих у технології Java. Хоча мова Java розроблена так, щоб її можна було інтерпретувати, технічно ніщо не заважає скомпілювати байт-код у виконуваний код. Динамічна компіляція застосовується до байт-коду, надісланого через мережу, але це жодним чином не впливає на переносимість і безпеку, оскільки програмою все ще керує система виконання. Цей підхід, який використовується в багатьох системах виконання Java, забезпечує оптимізовану продуктивність на рівні коду C++.

Мова Java є однією з наймолодших мов у сімействі мов програмування, і вона була розроблена з розрахунком, що професійні програмісти зможуть легко її освоїти та ефективно використовувати. Java заснована на синтаксисі C++ - безсумнівно, однієї з найпопулярніших сучасних мов програмування. Однак Java є абсолютно самостійною мовою програмування, на початку її народження не було проблем сумісності з C++. Отже, деякі механізми реалізовані в Java по-різному, а деякі – ні. Ідеологічно Java побудована інакше, ніж C++. розробник

Java базується на досвіді розробки програм на C++ і намагається позбутися функцій, які виявляються невизначеними. Як наслідок, у Java немає перевантаження операторів і автоматичного перетворення несумісних типів — конструкцій, які можуть бути важкорозпізнаним джерелом помилок, якщо вони використовуються неправильно. Загалом, інтерфейси Java простіші та

легші для розуміння. На Java набагато легше писати програми з графічним інтерфейсом. Звичайно, простота інтерфейсу компенсується меншою гнучкістю, і бібліотека Java не така багата, як стандартна бібліотека C/C++. Але пам’ятайте, що Java розроблена для використання на різних платформах, тому реалізовано найбільш стандартні функції, щоб полегшити її адаптацію до конкретного середовища.

Java успадкувала потужні механізми об'єктно-орієнтованого програмування від C++. Оскільки Java розробляється «в білому просторі», тобто без гарантії сумісності з попередніми версіями, розробники мають повну свободу думок. Результат – чіткий і прагматичний підхід до об’єктів. Вільно приймаючи ідеї, які були реалізовані протягом останніх кількох десятиліть, Java вдається знайти баланс між парадигмою «все є об’єктом» і прагматичним підходом. Об’єктна модель Java є простою та легко розширюваною, а типи запитів, наприклад Integer, зберігаються як необ’єктні дані, що може значно збільшити швидкість обробки.

У Java є вбудований набір ключів класів, які містять основні абстракції реального світу, з якими має працювати ваша програма. Популярність Java заснована на вбудованих абстрактних класах, які роблять її дійсно незалежною від платформи мовою.

Насправді більшість архітектурних рішень, прийнятих під час створення Java, були мотивовані бажанням забезпечити синтаксис, подібний до C та C++. Java використовує майже однакові вимоги до оголошення змінних, передачі параметрів, операторів і керування потоком виконання коду. Java додає всі переваги C++, але виключає недоліки останнього.

Покажчики пам'яті або адреси є найпотужнішими та найнебезпечнішими функціями C++. Причиною більшості помилок у сьогоднішньому коді є саме неправильне використання покажчиків. Наприклад, одна з типових помилок - неправильно розрахувати одну одиницю розміру масиву і знищити вміст комірки пам'яті після неї.

Хоча дескриптори об'єктів реалізовані як покажчики в Java, у ній відсутня можливість використовувати їх безпосередньо. Ви не можете перетворити ціле число на покажчик, а також не можете посилатися на довільну адресу пам’яті.

Однак у Java можна створювати не лише аплети, а й консольні програми, програми GUI, сервлети та JSP.

Незабаром після появи технології сервлетів розробники зіткнулися з такою проблемою: щоб динамічно генерувати HTML-сторінки за допомогою сервлетів, код HTML потрібно було розмістити в самому сервлеті. При цьому HTML-код сторінки змішується з кодом Java (тоді як логіка програми змішується із зовнішнім виглядом веб-сторінки), що ускладнює роботу як програмістів, так і веб-дизайнерів.

Для вирішення цієї проблеми була розроблена технологія JavaServer Pages (JSP). Це дозволяє розмістити код Java у HTML-коді веб-сторінки. Коли до сторінки jsp звертаються вперше, її код автоматично перетворюється на сервлет і компілюється. Згодом, під час наступних викликів, веб-сервер викликає не сторінку jsp, а скомпільований сервлет. Коли до сторінки jsp вносяться зміни, веб-сервер виявляє, що сторінка змінилася, і знову оновлює відповідний сервлет.

Поняття контейнера (контейнера) було введено в сервлет і технологію JSP. Контейнер сервлетів — це механізм, відповідальний за виконання сервлетів. Контейнер JSP — це механізм, відповідальний за перетворення сторінок jsp на сервлети та передачу цих сервлетів у контейнер Servlets. Оскільки сервлети та JSP-сторінки викликаються через протокол HTTP, контейнер зазвичай супроводжується іншим компонентом - веб-сервером. Набір веб-серверів і контейнерів утворює сервер веб-додатків (рис. 1).

Поєднання технологій Servlet і JSP з кількома іншими технологіями Java називається Java 2 Enterprise Edition (J2EE). Таким чином з'явилися сервери додатків Java у різних компаній (IBM, BEA, IONA, Borland), в тому числі і в

самій Sun. Кожна компанія реалізувала власну технологію J2EE на своєму сервері додатків, але всі вони відповідали специфікаціям Sun.
Sun раніше надавала безкоштовну довідкову реалізацію сервера веб-додатків Java під назвою JServ. Після випуску технології J2EE весь код був переданий Apache Software Foundation, а продукт отримав назву Tomcat.
Наразі Tomcat є еталонною реалізацією веб-сервера Java і є частиною групи проекту Apache під назвою Jakarta.

Мультиплатформенність Web-середовища висуває надзвичайно високі вимоги до надійності програми. Тому при розробці Java пріоритетним було вміння створювати відмовостійкі програми. Java звільняє програмістів від хвилювання про багато поширених причин програмних помилок. Як згадувалося раніше, Java — це строго типізована мова програмування. Система Java, яка все ще виконується, бере на себе «збирання сміття», тобто автоматично звільняє динамічно виділену пам’ять. Звичайно, це певною мірою знижує ефективність коду, але запобігає типовим помилкам, коли програміст забуває звільнити виділену пам’ять або, навпаки, звільняє пам’ять, яка все ще використовується. Java підтримує об'єктно-орієнтовану обробку винятків, подібну до C++. Але на відміну від C++ в Java, обробка винятків є обов’язковою. Тобто неможливо скомпілювати програму, яка відкриває файл без обробки помилок «файл не знайдено», які можуть виникнути під час відкриття файлу. Добре написана програма Java сама справляється з усіма помилками виконання.

Java була розроблена з акцентом на задоволення вимог для створення інтерактивних програм, які працюють з мережею. З цією метою Java підтримує багатозадачне програмування, що дозволяє легко розробляти програми, які породжують багато процесів одночасно. Виконання програми Java базується на елегантному, але в той же час високоорганізованому рішенні багатопроцесної синхронізації, яке дозволяє створювати ефективні інтерактивні системи.

У Java реалізовано кілька цікавих рішень, які дозволяють писати код, який виконує багато різних функцій одночасно, не забуваючи стежити за тим, що і коли має статися. У мові Java найелегантніший метод, винайдений на сьогоднішній день, застосований для вирішення проблеми синхронізації процесів, щоб можна було створювати прекрасні інтерактивні системи. Прості у використанні складні підпроцеси Java надають можливість реалізувати певну поведінку в програмі, не відволікаючись на створення глобального обробника подій циклу.

Основними проблемами, з якими стикаються розробники Java, є довговічність і портативність. Одна з головних проблем, з якою стикаються програмісти, полягає в тому, що немає гарантії, що програма, написана сьогодні, однаково успішно працюватиме на тій же машині завтра. Оновлення операційної системи, модернізація процесора та зміна обсягу оперативної пам'яті можуть привести до збою програми. Розробники Java спробували це змінити та прийняли кілька складних рішень щодо мови Java та виконання програм Java. Їхня мета – «одна робота скрізь, будь-коли та будь-де». Таким чином, Java є системою, яку можна легко розширити шляхом створення нових стандартних класів і бібліотек.

Як згадувалося раніше, Java дозволяє створювати незалежні від платформи програми, компілюючи їх у проміжне представлення, яке називається байт-кодом. Багато попередніх спроб вирішити проблему незалежності платформи робилися за рахунок продуктивності. Інтерпретовані системи, такі як BASIC і Perl, страждали від майже непереборних недоліків продуктивності. Java була створена з цією метою. Незважаючи на те, що Java є інтерпретованою мовою, генерація байт-коду ретельно оптимізована, щоб отриманий байт-код можна було легко перетворити на машинний код, який працює з дуже високою продуктивністю. Ця система виконання не втрачає жодних переваг портативного коду.

Мова Java призначена для створення програм, які працюють у розподіленому середовищі Інтернет на основі протоколу TCP/IP. Насправді

доступ до ресурсу за допомогою URL-адреси не те саме, що доступ до файлу. Крім того, Java має спосіб транспортування повідомлень у внутрішньому адресному просторі. Це дозволяє дистанційно виконувати процес. Ці інтерфейси містяться в пакеті RMI (Remote Method Invocation). Цей інструмент забезпечує високий рівень абстракції для програмування в середовищах клієнт/сервер.

Програма Java містить велику кількість інформації про тип виконання, яка використовується для надання доступу до об’єктів під час виконання програми. Це забезпечує безпечне та оптимальне динамічне розташування. Таким чином реалізується безпека середовища виконання малих програм [8].

Однією з найбільших переваг мови Java є можливість створювати аплети, невеликі програми, які запускаються у веб-браузерах. Однак, оскільки вони виконуються на комп'ютері користувача, на аплети накладаються певні обмеження:

- аплет не може отримати доступ до жорсткого диска. Однак для них існує система цифрового підпису, за допомогою якої користувачі можуть бути впевнені, що ці аплети надходять із надійного джерела, таким чином усуваючи більшість обмежень
- аплету потрібен деякий час для завантаження з Інтернету. Щоб скоротити цей час, усі дані, необхідні для роботи аплету, зазвичай містяться в архіві jar, що дозволяє завантажувати аплети набагато швидше.

Перевагами аплетів є:

- для аплетів немає необхідності встановлювати їх, як інші програми. Використовуйте його, коли вам потрібно постійно завантажувати оновлені версії програми;
- не потрібно турбуватися про те, що завантажені аплети виконують потенційно небезпечні операції. Заборонено виконувати базові операції з аплетами, які можуть спричинити втрату, пошкодження важливої інформації або змінити вміст файлу.

В мові *Java* існує два способи створити аплет. Створити підклас суперкласу *Applet* (визначений в пакеті *java.applet.Applet*) або створити підклас суперкласу *Japplet* (визначений в пакеті *javax.swing*).

Різниця між цими двома класами полягає лише в тому, що перший був визначений попередньою версією *Java*, яка не підтримує повну архітектурну незалежність, а графічний інтерфейс не дуже гарний. Другий був створений у пізнішій версії *Java*, підтримує незалежність від архітектури та має кращий зовнішній вигляд *GUI*. Другий спосіб створення аплетів сьогодні в основному використовується для створення аплетів.

На відміну від консольних програм, які виконуються без опитування подій користувача, аплети очікують певних подій від користувача та змінюють весь стан на основі цих дій. Він просто «зависає» в пам'яті комп'ютера і перевіряє події, що відбуваються (рухи миші, натискання кнопок і клавіш тощо), а коли отримує операцію перевантаження, змінює свій стан.(рис. 2.1)

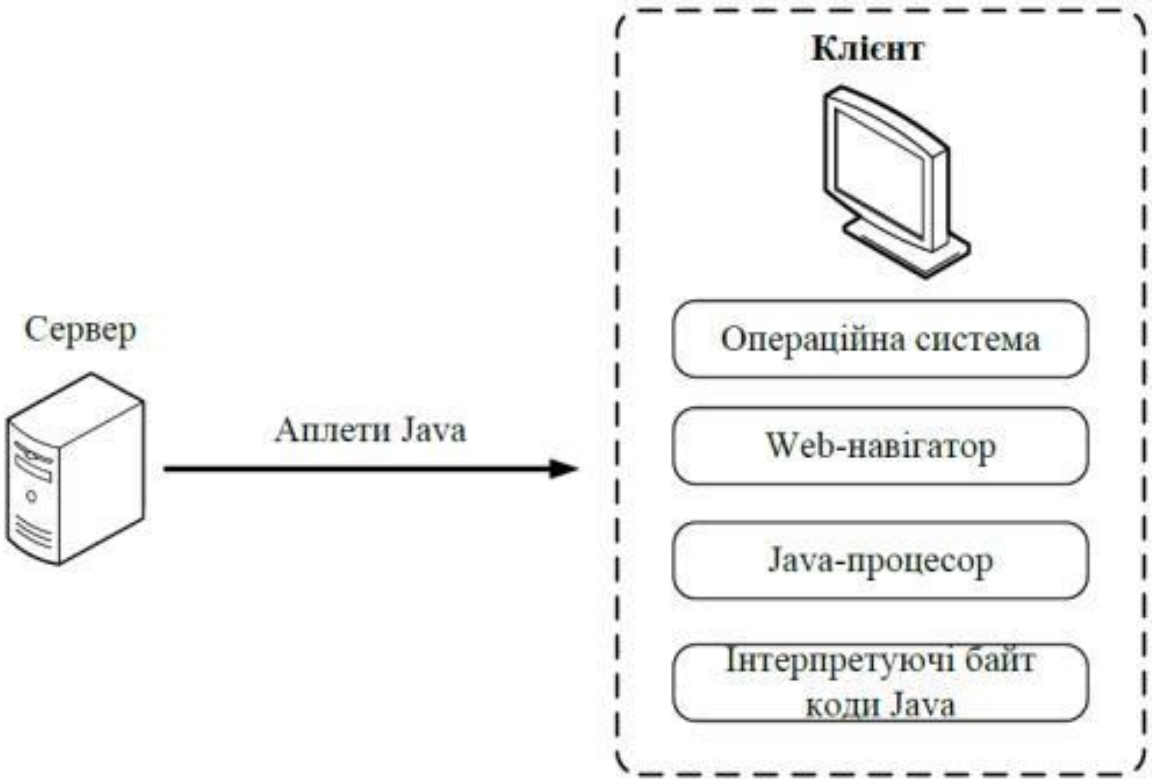


Рис.2.1 Робота аплетів

Для керування розміщенням елементів у вікнах аплетів використовується система менеджера розміщення. У *Java* існує кілька типів менеджерів розміщення. Коротко опишемо властивості деяких з них.

FloyLayout – розташовує компоненти в порядку зліва направо, якщо вони розміщені в одному рядку. Потім він переходить до наступного рядка тощо.

GridLayout – представляє вікна аплетів у вигляді таблиці *N*M*, де числа N і M вказуються під час створення менеджера. Після цього він розміщує елементи аплету в комірці так само, як і попередній. Різниця полягає в тому, що ви можете задати кожному елементу свою позицію в таблиці.

BolderLayout - для полярного розташування елементів. Він визначає значення, що відповідають сторонам світу. Таким чином, ми можемо розміщувати елементи з певним вирівнюванням.

CardLayout - компонент, призначений для макета блокнота. Усі елементи, що містяться в контейнері, визначаються як сторінки блокнота, тому в будь-який момент часу відображається лише одна сторінка.

GridBagLayout — це найзагальніший спосіб розміщення компонентів. Менеджер розглядає контейнер як таблицю, де кожен компонент може займати кілька клітинок. Для керування розміщенням елементів у вікнах аплетів використовується система менеджера розміщення. У Java існує кілька типів менеджерів розміщення. Коротко опишемо властивості деяких з них.

FloyLayout – розташовує компоненти в порядку зліва направо, якщо вони розміщені в одному рядку. Потім він переходить до наступного рядка тощо.

GridLayout – представляє вікна аплетів у вигляді таблиці N*M, де числа N і M вказуються під час створення менеджера. Після цього він розміщує елементи аплету в комірці так само, як і попередній. Різниця полягає в тому, що ви можете задати кожному елементу свою позицію в таблиці.

BolderLayout - для полярного розташування елементів. Він визначає значення, що відповідають сторонам світу. Таким чином, ми можемо розміщувати елементи з певним вирівнюванням.

CardLayout - компонент, призначений для макета блокнота. Усі елементи, що містяться в контейнері, визначаються як сторінки блокнота, тому в будь-який момент часу відображається лише одна сторінка.

GridBagLayout — це найзагальніший спосіб розміщення компонентів. Менеджер розглядає контейнер як таблицю, де кожен компонент може займати кілька клітинок.

Взаємодія між аплетами може базуватися на двох методах:

- створення каналів між аплетами (PipeInputStream/PipeOutputStream);
- використання сокетів (якщо це окремі програми);
- використання елементів сторінки HTML через параметри аплету.

Вищезазначена простота програмування на Java є причиною того, що розробка на Java є менш дорогою, ніж аналогічна розробка на потужніших мовах програмування. Портативність Java-програм також сприяє цьому, оскільки виключаються витрати, пов'язані з адаптацією програм до конкретної платформи. Крім того, інтегрована оболонка для розробки програм на Java набагато дешевша ($70-100), ніж аналогічний C++ продукт Delphi (~$1000). Набір інструментів JDK (Java Development Kit) для пакетної компіляції програм Java, як правило, є безкоштовним програмним забезпеченням. Тому платформу Java можна рекомендувати як ідеальну платформу для створення некомерційних програмних продуктів, особливо для освітньої сфери. [6]

Оскільки як говорилось вище мова *Java* чисто об’єктно-орієнтована мова не можна оминути цей момент.

Розглянемо особливості підходу об'єктно-орієнтованого програмування в порівнянні з функціональним підходом. Нагадаємо, що систематику методів програмування ми встановили на вступних лекціях.

Найважливішим кроком на шляху до вдосконалення мов програмування стала поява об'єктно-орієнтованого програмування (або скорочено ООП) і відповідних класів мов. Вивчення теорії та практики проектування та

реалізації програмних систем відповідно до принципів ООП є основною метою другої частини курсу.

В об’єктно-орієнтованому підході програма — це опис об’єктів, їхніх атрибутів (або властивостей), колекцій (або класів), зв’язків між ними, методів, за допомогою яких вони взаємодіють, і операцій (або методів) супероб’єктів.

Беззаперечною перевагою такого підходу є концептуальна наближеність до предметних галузей довільної структури та призначення. Механізм успадкування властивостей і методів дозволяє створювати похідні концепції поверх базових концепцій, створюючи модель будь-якої складної предметної області з заданою властивістю.

Іншою теоретично цікавою і практично важливою особливістю об'єктно-орієнтованого підходу є підтримка механізмів обробки подій, які змінюють властивості об'єктів і імітують їх взаємодію в предметній області.

Уздовж ієрархії класів від загальних концепцій предметної області до більш конкретних концепцій (або від більш складних до більш простих концепцій) і навпаки, програміст має можливість змінювати рівень абстракції або конкретності свого погляду на реальний світ, яким він є. робить модель.
Використання раніше розроблених (можливо, розроблених командами інших програмістів) бібліотек об'єктів і методів дозволяє заощадити значні трудовитрати при виробництві програмного забезпечення, особливо стандартного програмного забезпечення.

Об'єкти, класи та методи можуть бути поліморфними, що робить реалізоване програмне забезпечення більш гнучким та універсальним.
Складність адекватної (безсуперечної та повної) формалізації теорії об’єктів створює труднощі при тестуванні та валідації створеного програмного забезпечення. Мабуть, така ситуація є одним з найбільш істотних недоліків підходу об'єктно-орієнтованого програмування.

Найвідомішим прикладом об'єктно-орієнтованої мови програмування є мова *C++*, яка була розроблена з імперативної мови C. Його прямим нащадком

і логічним продовженням є мова C#, що вивчається в цьому курсі. Інші приклади об'єктно-орієнтованих мов програмування: *Visual Basic, Java, Eiffel, Oberon*.

Перехід від структурного процедурного підходу до об’єктно-орієнтованого програмування, як і перехід від мови програмування низького рівня до мови високого рівня, вимагає значних витрат на навчання. Природно, це відбувається за рахунок збільшення продуктивності програміста при розробці та впровадженні програмного забезпечення. Ще однією перевагою ООП над імперативними підходами є більш широке повторне використання розробленого програмного коду. У той же час, на відміну від попередніх підходів до програмування, об'єктно-орієнтований підхід вимагає глибокого розуміння основ, або іншими словами, концепцій, на яких він базується. Основні поняття ООП зазвичай включають абстракцію даних, успадкування, інкапсуляцію та поліморфізм.

У більшості випадків під час практики програмування та навчальних курсів студенти не мають чіткої математичної основи для розробки достатньо повної та чіткої концепції основ ООП. Сильна сторона запропонованого курсу полягає в тому, що частини інформатики, які вже розглянуті в першій частині курсу (наприклад, лямбда-числення та комбінаторна логіка), дозволяють розвинути глибоке та точне розуміння фундаментальних концепцій об’єктно-орієнтованого програмування. Зокрема, концепція абстрактного - Основна операція лямбда-числення - вже знайома нам.

Давайте якісно пояснимо основи ООП. Успадкування конкретних властивостей об’єкта та функцій роботи об’єкта базується на ієрархічній структурі, а інкапсуляція полягає в тому, щоб «приховати» властивості та методи всередині об’єкта. Як і функціональне програмування, поліморфізм розуміється як наявність функцій зі здатністю обробляти дані змінних типів.

Розглянемо докладніше фундаментальні принципи підходу об'єктно-орієнтованого програмування як абстракції.

У розділі математики, який вивчає моделювання процесу створення програм, абстракція розуміється як довільне вираження мови програмування, відмінне від ідентифікаторів.

Найважливіші операції, які ми вивчали в першій частині курсу, це ті, які обчислюють значення виразу або команди, тобто нотаційні операції (зокрема, функції, які обчислюють значення, явно створюються з використанням семантики мови програмування). У зв'язку з цим важливо визначити, в чому полягає сенс абстракції. Будемо вважати, що значення функції або змінної можна присвоїти абстракції і є значенням останньої.

В об'єктно-орієнтованому програмуванні кожен об'єкт в основному є динамічною сутністю, тобто змінюється з часом (і під впливом зовнішніх факторів). Іншими словами, об'єкти мають той чи інший спосіб поведінки. Стосовно абстракцій, які є об’єктами, поведінка складається з приєднання функцій до параметрів.

Як ми вже зазначали, абстракції в об'єктно-орієнтованому програмуванні можна адекватно моделювати за допомогою лямбда-числення. Точніше, абстрактні операції – це саме модель однойменної концепції ООП.

Іншою фундаментальною концепцією об'єктно-орієнтованого програмування є наслідування цієї інтуїтивно зрозумілої концепції. У неформальній обстановці імітація відноситься до властивостей того чи іншого об'єкта, які є похідними від деякого базового об'єкта, щоб зберегти поведінку (тобто властивості та операції над ними), характерну для батьківського об'єкта. З точки зору мови програмування, концепція успадкування означає застосування до всіх або Лише певні властивості або методи базового класу (або батьківського класу) застосовуються до всіх похідних від нього класів. Крім того, ви повинні переконатися, що властивості та/або методи базового класу збережені для всіх екземплярів (тобто конкретних об’єктів) будь-якого похідного класу. У математиці успадковані поняття часто моделюються, наприклад, частково впорядковані установки (що є свого роду ієрархією).

Концепція імітації повністю формалізована математично за допомогою однієї з наступних нотацій:

- символ рамки Руссопулоса (названий на честь його творця Н. Д. Руссопулоса);
- діаграма Хассе (названа на честь Х. Хассе, вченого, який першим запропонував це візуальне представлення генетики).

ВИСНОВКИ ДО РОЗДІЛУ 2

У цьому розділі ми визначаємо основні алгоритми оптимізації такі як генетичний, світлячків, бур'янистої оптимізації, зозулин пошук та інші. Проводимо порівняльну характеристику алгоитмів та приходимо до висновку що в нашому випадку кращим варіантом буде генетичний алгоритм. Його перевагами є:

1. Легко модифікується.
2. Підтримка багатоцільової оптимізації.
3. Легко паралелізується.
4. Використовують правила імовірнісного переходу.
5. Надійний до локальних мінімумів / максимумів.

А з недоліків лише:

1. Обчислювально дорогий.
2. Будь-який невідповідний параметр ускладнить сходження алгоритму або просто дасть безглузді результати.

Серед двох недолікі варто розуміти що перший нівелюється правильними налаштуваннями та паралелізмом, а другий є характерним майже для кожного алгоритму. В свою чергу преваги доволі важливі адже паралелізм дозволить прискорити роботу, а легкість модифікації дозволить швидко змінити ціль роботи алгоритму.

Також в даному розділі ми розглянули можливості мови Java, та обрали її для розробки системи.

РОЗДІЛ 3. РОЗРОБКА ТА ТЕСТУВАННЯ СИСТЕМИ

3.1. Принципи реалізації генетичного алгоритму

Генетичні алгоритми засновані на моделюванні популяцій особин протягом великої кількості поколінь. Під час роботи генетичного алгоритму з'являються нові особини з кращими параметрами, а найменш успішні особини гинуть. Для ясності нижче наведено терміни, які використовуються в генетичних алгоритмах. Особина - одне значення x серед безлічі можливих значень разом зі значенням цільової функції для даної точки x.

1. Хромосоми - значення x.
2. Хромосома - значення x і в разі, якщо x є вектором.
3. Функція пристосованості (фітнес-функція, цільова функція) - оптимізуюча функція $f(x)$.
4. Популяція - сукупність особин.
5. Покоління – ці номер порядку створення популяції в генетичному алгоритмі.

Генетичні алгоритми засновані на моделюванні популяцій особин протягом великої кількості поколінь. Під час роботи генетичного алгоритму з'являються нові особини з кращими параметрами, а найменш успішні особини гинуть. Для ясності нижче наведено терміни, які використовуються в генетичних алгоритмах. [18].

Структурна схема генетичного алгоритму показана на рисунку 3.1:

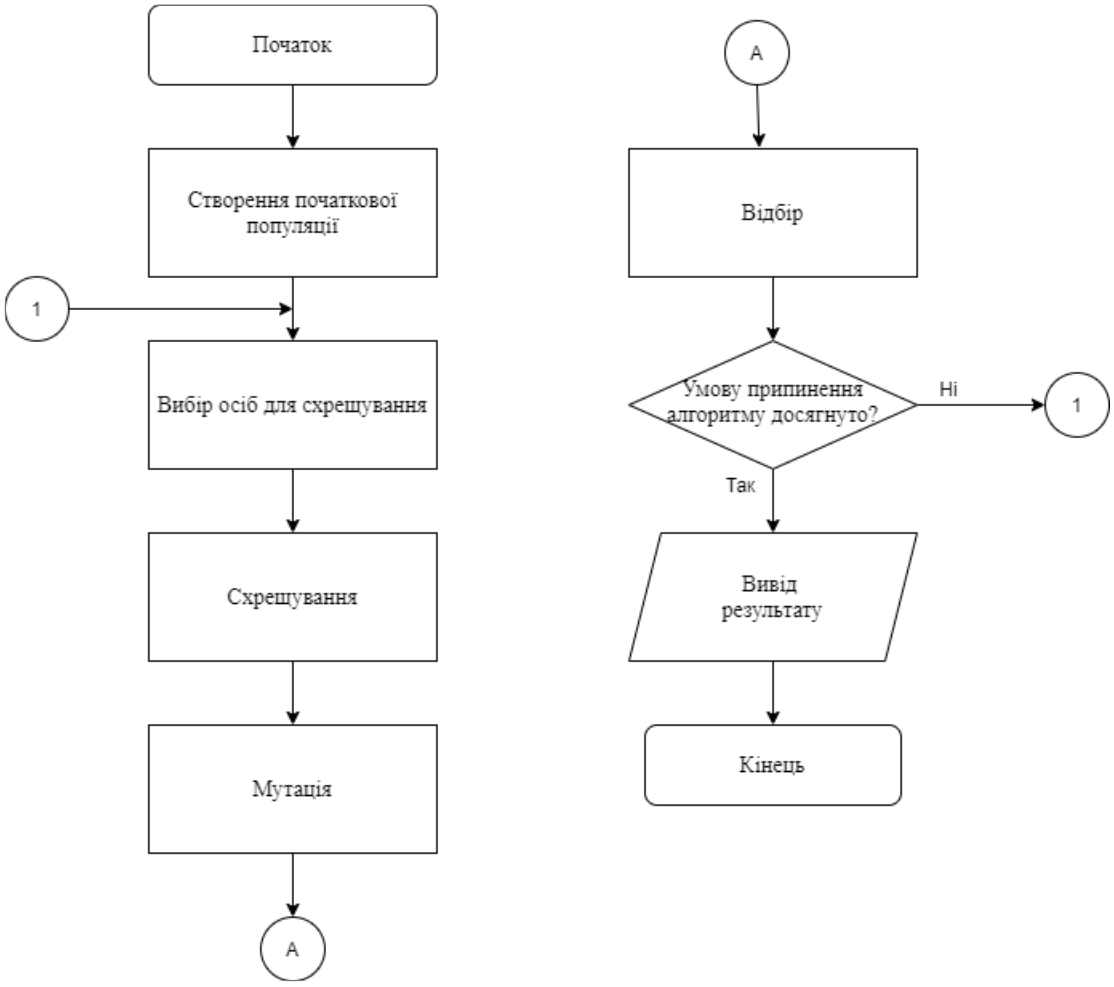


Рис. 3.1. Блок-схема генетичного алгоритму

Розглянемо кожен етап алгоритму більш детально.

Перший етап роботи алгоритму - це створення початкової популяції, тобто створення безлічі особин з різними значеннями хромосом x. Як правило, початкову популяцію створюють з особин з випадковим значенням хромосом, при цьому намагаються, щоб значення хромосом в популяції покривали всю область пошуку рішення відносно рівномірно, якщо немає якихось припущень щодо того, де може перебувати глобальний екстремум.

Чим більше особин буде створено на даному етапі, тим більша ймовірність того, що алгоритм знайде правильне рішення, а також при збільшенні розміру початкової популяції як правило потрібна менша кількість ітерацій генетичного алгоритму (кількість поколінь). Однак при зростанні розміру популяції потрібна все більша кількість розрахунків цільової функції і виконання інших генетичних операцій на наступних етапах алгоритму. Тобто

при збільшенні розміру популяції збільшується час розрахунку одного покоління.

В принципі, розмір популяції не зобов'язаний залишатися незмінним протягом усієї роботи генетичного алгоритму, часто зі збільшенням номера покоління розмір популяції можна зменшувати, тому що згодом все більша кількість особин будуть розташовуватися все ближче до шуканого рішення. Однак часто розмір популяції підтримують приблизно постійним.

Другим етапом є вибір особин для схрещування.Після створення популяції потрібно визначити, які особи будуть брати участь в операції схрещування. Для даного етапу існують різні алгоритми. Найпростіший з них - схрещувати кожну особину з усіма, але тоді на наступному етапі доведеться виконувати дуже багато операцій схрещування і розрахунку значень цільової функції. У цій роботі, я є тим хто визначаю парамктр вірогідності схрещування особин, схрещуватись будуть тільки ті особини, які входять в вірогідний проміжок, а інші особини позбавляються можливості залишати потомство.

Таким чином, на даному етапі потрібно створити пари особин, для яких на наступному етапі буде проводитися операція схрещування.

Результатом даної операції буде список партнерів для схрещування.

Наступним і одним з найважливіших етапів виступає схрещування особин. Схрещування - це генетична операція, в результаті якої створюються нові особини з новими значеннями хромосом. Нові хромосоми створюються на основі хромосом батьків. Найчастіше в процесі схрещування одного набору партнерів створюється одна дочірня особина, але теоретично дочірніх особин може створюватися більше. Алгоритм схрещування також можна реалізувати різними способами.

Сенс схрещування полягає в тому, що беремо одну частину хромосоми і другу частину іншого виду і створюємо з них нову хромосому, яка і буде міститися в новому вигляді. Часто хромосоми схрещують побітово. Суть його полягає в тому, що спочатку випадково вибираємо точку розриву

(схрещування) хромосоми, потім створюємо нову хромосому, яка складається з лівої частини першої хромосоми і правої частини другої.

Нехай, наприклад, у нас є дві хромосоми (8-бітові для простоти): 10110111 і 01110010. Випадково вибираємо точку розриву (відзначена символом '|'):

101 | 10111

011 | 10010

Звідси ми може зробити дві різні хромосоми це 101 10010 і 011 10111. Яку з них вибрати - це ми вирішуємо генератором випадкових чисел. Також можна шукати дві і більше точок перетину.

У цій роботі я схрещував хромосоми типу Double, тому в даному випадку в основному є дві точки перетину - посередині слова і цифрових символів, тобто спочатку хромосоми схрещуються порозрядно і не враховуючи знак, тоді ми також випадково беремо знак з однієї з хромосом.

Після кросинговеру логічно слідує не менш важлива фаза алгоритму — мутація. Мутація є важливою стадією генетичного алгоритму, яка підтримує різноманітність окремих хромосом, тим самим зменшуючи ймовірність швидкого зближення рішення до деякого локального мінімуму замість глобального мінімуму. Мутації — це випадкові зміни окремих хромосом, щойно отримані шляхом схрещування.

Зазвичай ймовірність мутації не встановлюється дуже високою, щоб мутація не заважала збіжності алгоритму, інакше вона знищить особин з хорошими хромосомами.

Окрім кросинговеру, для мутації також можна використовувати різні алгоритми. Наприклад, ви можете замінити одну або кілька хромосом на випадкові значення. У даній роботі використовується побітова мутація, коли в хромосомі інвертується один біт, як показано на Рис 3.2.

Рис. 3.2 – Мутація одного біта

Особи можуть стати більш-менш успішними в результаті мутацій, але ця маніпуляція допускає ненульову ймовірність успішної хромосоми, що містить нульові набори та нульові набори, відсутні в батьківській особині.

В результаті гібридизації і подальшої мутації виникають нові особини. Якщо не вжити необхідних заходів для зменшення чисельності популяції, кількість особин зростатиме експоненціально, що потребуватиме все більше оперативної пам'яті та часу для обробки кожного нового покоління популяції. Тому після появи нових особин необхідно вилучати з популяції найменш успішних особин. Це досягається за рахунок правильного відбору особин за спеціальними алгоритмами.

Алгоритм підбору також може бути різним. Як правило, першими відкидають особини, чиї хромосоми не потрапляють у заданий інтервал для пошуку рішення. Крім того, у багатьох реалізаціях я сортую види відповідно до значення цільової функції, тобто зберігаю найбільш «жваві» види.

Нарешті, шостий етап — кінець алгоритму, який завжди позначається певною умовою — критерієм кінця алгоритму. Критерієм завершення алгоритму є вказівка певної кількості ітерацій (алгебра). Але цей критерій слід використовувати з обережністю, оскільки генетичні алгоритми є імовірнісними, і різні цикли алгоритму можуть сходитися з різною швидкістю. Тому в якості порогу кількості ітерацій необхідно встановити досить велике число.

1.2 Розробка системи для вирішення задачі дискретної оптимізації

Важливо зазначити, що в даній системі велику кількість грає бібліотека *java.util.stream*, яка дозволила спростити розробку та покращити роботу програми. Тому вважаю що її варто розглянути детальніше.

Ключовою абстракцією, представленою в цьому пакеті, є потік . Класи *Stream, IntStream, LongStream,* і *DoubleStream* є потоками над об'єктами та примітивними *int, longi doubleмипами*. Потоки відрізняються від колекцій кількома ознаками:

- Без зберігання. Потік — це не структура даних, яка зберігає елементи; натомість він передає елементи з джерела, такі як структура даних, масив, функція генератора або канал введення/виведення, через конвеєр обчислювальних операцій.
- Функціональний характер. Операція над потоком створює результат, але не змінює його джерело. Наприклад, фільтрування *Stream* отриманого з колекції створює новий *Stream* без фільтрованих елементів замість видалення елементів із вихідної колекції.
- Прагнення до ліні. Багато потокових операцій, таких як фільтрація, зіставлення або видалення дублікатів, можуть бути реалізовані ліниво, відкриваючи можливості для оптимізації. Наприклад, «знайти першу *String* з трьома послідовними голосними» не потрібно перевіряти всі вхідні рядки. Потокові операції поділяються на проміжні операції та термінальні (створення вартості або побічного ефекту) операції. Проміжні операції завжди ліниві.
- Можливо необмежений. Хоча колекції мають кінцевий розмір, потоки не потребують. Операції короткого замикання, такі *як limit(n)* або *findFirst()* можуть дозволити завершити обчислення нескінченних потоків за кінцевий час.
- Витратний матеріал. Елементи потоку відвідуються лише один раз протягом життя потоку. *Як* і *Iterator*, новий потік повинен бути згенерований для повторного перегляду тих же елементів джерела.

Потоки можна отримати кількома способами. Деякі приклади:

- Від а *Collectionчерез stream()* і *parallelStream()*методи;
- З масиву через *Arrays.stream(Object[])*;
- Зі статичних фабричних методів у класах потоку, таких як *Stream.of(Object[]), IntStream.range(int, int) або Stream.iterate(Object, UnaryOperator)*;
- Рядки файлу можна отримати з *BufferedReader.lines()*;
- Потоки шляхів до файлів можна отримати з методів у *Files*;

- Потоки випадкових чисел можна отримати з *Random.ints()*;
- Численні інші методи передачі потоку в *JDK*, включаючи *BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence)і Jar File.stream()*.

Додаткові джерела потоку можуть бути надані сторонніми бібліотеками за допомогою цих методів .[20]

Потокові операції поділяються на проміжні та кінцеві та об'єднуються для формування потокових конвеєрів . Конвеєр потоку складається з джерела (наприклад *Collection*, масиву, функції генератора або каналу введення/виведення); за якими слідує нуль або більше проміжних операцій, таких як *Stream.filterабо Stream.map*; і термінальна операція, така як *Stream.forEachабо Stream.reduce*.

Проміжні операції повертають новий потік. Вони завжди ліниві ; виконання проміжної операції, такої як *filter()*фактично не виконує жодної фільтрації, а натомість створює новий потік, який під час обходу містить елементи початкового потоку, які відповідають даному предикату. Обхід джерела конвеєра не починається, доки не буде виконана кінцева операція конвеєра.

Термінальні операції, такі як *Stream.forEachабо IntStream.sum*, можуть проходити через потік, щоб створити результат або побічний ефект. Після виконання термінальної операції потоковий трубопровід вважається використаним і більше не може використовуватися; якщо вам потрібно знову пройти те саме джерело даних, ви повинні повернутися до джерела даних, щоб отримати новий потік. Майже у всіх випадках термінальні операції працюють активно , завершуючи обхід джерела даних і обробку конвеєра перед поверненням. Тільки термінальні операції *iterator()* і *spliterator()*ні; вони надаються як «аварійний люк» для довільного проходження конвеєра, керованого клієнтом, у випадку, якщо існуючих операцій недостатньо для виконання завдання.

Лінива обробка потоків забезпечує значну ефективність у конвеєрі, такому як наведений вище приклад *filter-map-sum*, фільтрація, відображення та підсумовування можуть бути об’єднані в один прохід даних з мінімальним проміжним станом. Лінощі також дозволяють уникати перевірки всіх даних, коли це не потрібно; для таких операцій, як «знайти перший рядок, довжина якого перевищує 1000 символів», необхідно перевірити лише стільки рядків, щоб знайти той, який має потрібні характеристики, не перевіряючи всі рядки, доступні з джерела.

Проміжні операції далі поділяються на операції без стану та операції без стану. Операції без стану, такі як *filter* і *map*, не зберігають жодного стану попереднього елемента під час обробки нового елемента — кожен елемент може оброблятися незалежно від операцій над іншими елементами. Операції зі збереженням стану, такі як *distincti sorted*, можуть включати стан із раніше побачених елементів під час обробки нових елементів.

Операціям із збереженням стану може знадобитися обробити весь вхід, перш ніж отримати результат. Наприклад, неможливо отримати жодних результатів від сортування потоку, доки не побачите всі елементи потоку. Як наслідок, під час паралельного обчислення деякі конвеєри, що містять проміжні операції зі збереженням стану, можуть вимагати багаторазових проходів даних або буферизації значних даних. Конвеєри, що містять виключно проміжні операції без стану, можуть бути оброблені за один прохід, послідовний або паралельний, з мінімальною буферизацією даних.

Крім того, деякі операції вважаються операціями короткого замикання . Проміжна операція є короткозамикаючою, якщо, будучи поданою з нескінченним введенням, вона може створити кінцевий потік як результат. Термінальна операція є короткозамикаючою, якщо, будучи поданою з нескінченним введенням, вона може завершитися через кінцевий час. Наявність операції короткого замикання в конвеєрі є необхідною, але недостатньою умовою для нормального завершення обробки нескінченного потоку за кінцевий час.

В якості прикладу задачі дискретної оптимізації використаємо задачу формування розкладу. За принципом рішення дана задача є аналогічною класичній задачі оптимізації складу. Суть вирішення задачі полягає в максимально евективному розподілу занять, враховуючи всі характеристики кожного об’єкту що впливає на складання розкладу.

Рішення даної задачі розділемо на дві теки:

- domain – тека де описані об’єкти почакових даних що отримає алгоритм
- operation – тека де описані всіх операції що буде виконувати система.

3.2.1 Розробка класів об’єктів

В даній системі було створено шість класів, що описують відповідні об’єкти:

1. *Room.java* – представляє аудиторію в якій може проводитися заняття. На рис 3.1 зображено даний клас на діаграмі класів і з нього можна зрозуміти, що даний клас має два параметри: номер аудиторії і максимальну наповнюваність даної аудиторії

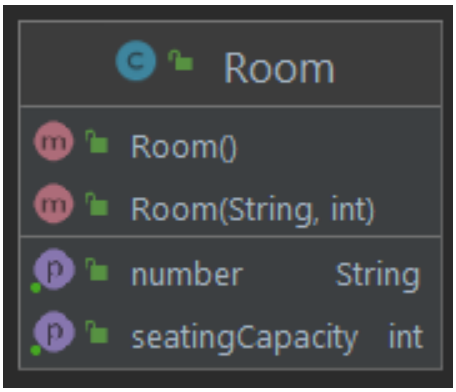


Рис. 3.1. Клас Room на діаграмі класів

2. *Instructor.java*, зображений на рис 3.2 представляє, викладача що проводитиме заняття. Аналогічно попередньому класові має два параметри: ім’я та *id* на випадок, якщо декілька викладачів матимуть одинакове ім’я.

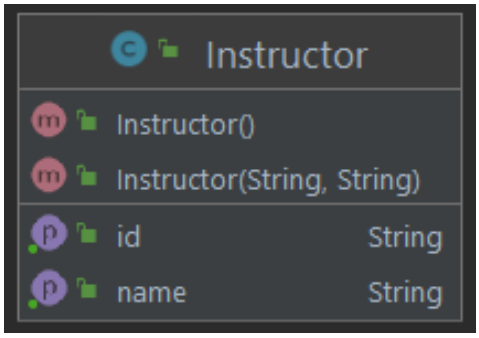


Рис. 3.2. Клас Instructor

3. *MeetingTime.java* – це клас що відповідає за час проведення заняття і має два поля: *time* – час і день проведення та *id* для додаткової нумерації(рис. 3.3).

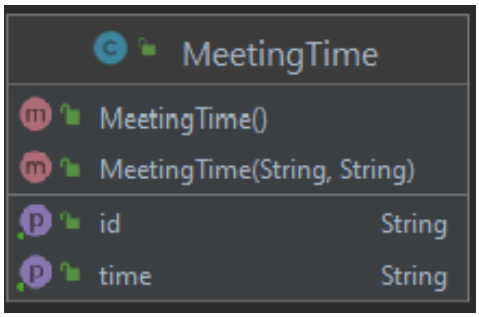


Рис. 3.3. Клас MeetingTime

4. *Department.java* – клас, що описує кафедру та предмети, що дана кафедра пропонує. Для збереження курсів використовується масив *ArrayList* що дозволяє динамічно змінювати курси за потреби та не накладає обмеження на кількість курсів для кафедри (рис. 3.4).

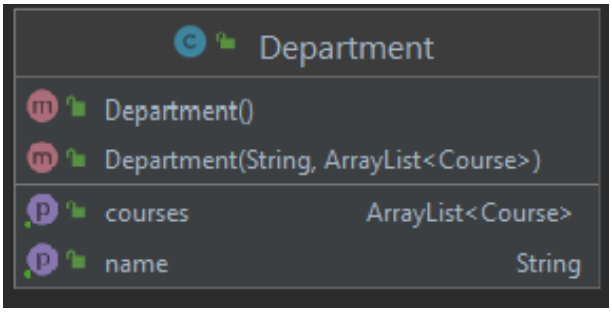


Рис. 3.4. Клас *Department*

5. *Course.java* – це клас, що представляє предмет і його характеристики. Даний клас має чотири змінні: номер курсу, код курсу, максимально можлива кількість відвідувачів цього курсу та список викладачів що можуть викладати даний предмет(рис. 3.5).

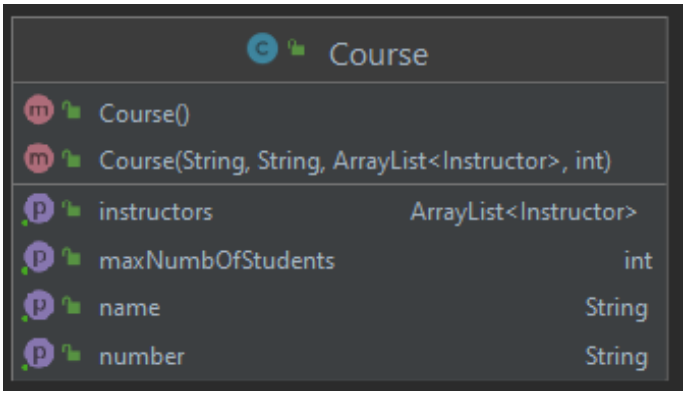


Рис. 3.5. Клас *Course*

6. *Class.java* – це клас, що представляє позицію в розкладі, тобто: кабінет, час проведення, викладача і предмет. Він використовує як свої праметри дані усіх попередніх класів, плюс власний ідентифікаційниий номер.(рис. 3.6)

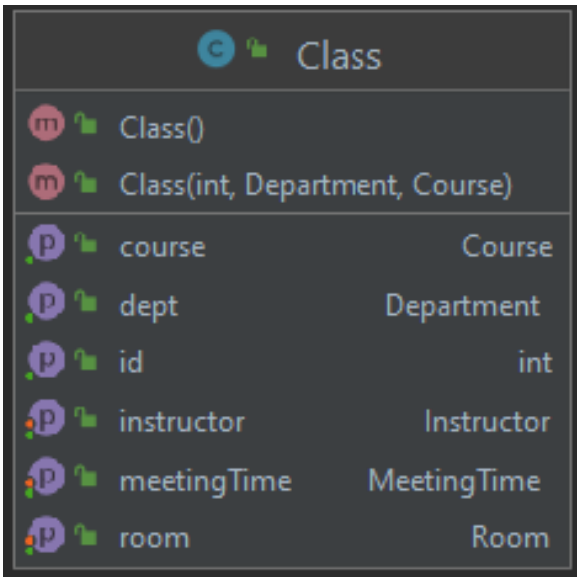


Рис. 3.6. Клас *Class*

Крім вище перечислених змінних, кожен клас має набір функції для повернення та встановлення значень змінних.

3.2.2. Розробка класів операцій

Також в даній системі було створено шість класів, що відповідають за побудову розкладу з допомогою генетичного алгоритму.

Першим класом що буде розглянуто буде клас *Data*. Цей клас зберігає всі доступні програмі дані та надає доступ до них програмі. Він зберігає дані в масивах *ArrayList* що дозволяє зберігати вклику кількість даних і ефективно їх редагувати за потреби.(рис. 3.7,3.8)

```
public class Data {
    private ArrayList<Room> rooms;  //list of rooms ava
    private ArrayList<Instructor> instructors;  //list
    private ArrayList<Course> courses;  //list of cours
    private ArrayList<Department> depts;    //list of d
    private ArrayList<MeetingTime> meetingTimes;    //l
    private int numberOfClasses = 0;    //total number
```

Рис.3.7 Реалізація класу *Data*

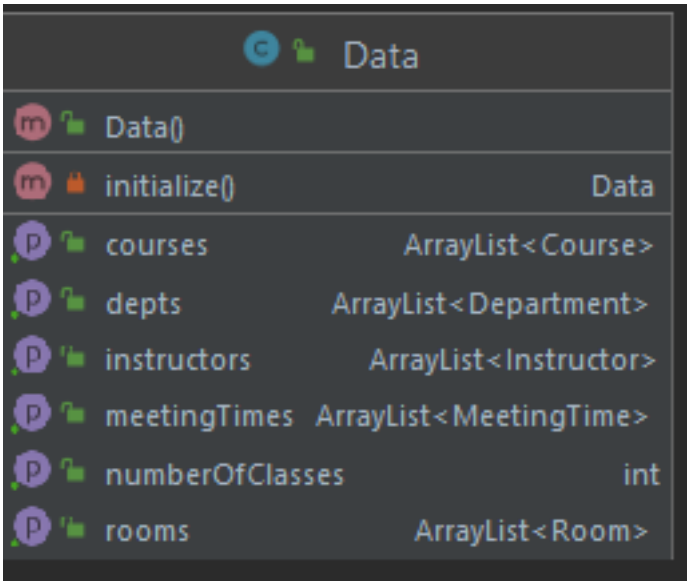


Рис.3.8 UML-діаграма класу *Data*

Наступним є клас *Population* який на основі даних попереднього класу створює першу популяцію і сортує її за придатністю(рис. 3.9)

```
public class Population {
    private ArrayList<Schedule> schedules;  //one schedule stores a list of classes

    public Population(int size, Data data) {
        schedules = new ArrayList<Schedule>(size);
        IntStream.range(0, size).forEach(x -> schedules.add(new Schedule(data).initialize()));
    }

    public ArrayList<Schedule> getSchedules() { return this.schedules; }

    public Population sortbyFitness() { //sorting in decreasing order
        schedules.sort((schedule1, schedule2) -> {
            int returnValue = 0;
            if(schedule1.getFitness() > schedule2.getFitness())
                returnValue = -1;
            else if(schedule1.getFitness() < schedule2.getFitness())
                returnValue = 1;
            return returnValue;
        });
        return this;
    }
```

Рис.3.9 Реалізація класу *Population*

Отримавши першу популяцію потрібно провести схрещування і мутацію задля отримання наступного покоління. Для цього в роботі був використаний клас *GeneticAlgorithm* що наслідує інтерфейс *Evolution.* Даний крок був зроблений для покращення зручності розробки і більш стабільної поведінки системи. Інтерфейс *Evolution* оголошує основні прототипи методів що надалі реалізуютьмя класом *GeneticAlgorithm.*(рис. 3.10)

```
public interface Evolution { //interface for methods of genetic algorithm
    Population evolve(Population population);
    Population crossoverPopulation(Population population);
    Schedule crossoverSchedule(Schedule schedule1, Schedule schedule2);
    Population mutatePopulation(Population population);
    Schedule mutateSchedule(Schedule mutateSchedule);
    Population selectTournamentPopulation(Population population);
}
```

Рис.3.10 Реалізація інтерфейсу *Evolution*

GeneticAlgorithm проводит схрещування перебираючи всі види і схрещуємо їх із установленою ймовірністю схрещування з іншими видами, які вибираються випадково.(рис. 3.11)

```
public class GeneticAlgorithm implements Evolution{
    private Data data;

    public GeneticAlgorithm(Data data) { this.data = data; }

    public Population evolve(Population population) { return mutatePopulation(crossoverPopulation(population)); }
    public Population crossoverPopulation(Population population) {
        Population crossoverPopulation = new Population(population.getSchedules().size(), data);
        IntStream.range(0, Driver.NUMB_OF_ELITE_SCHEDULES).forEach(x -> crossoverPopulation.getSchedules().set(x,
        IntStream.range(Driver.NUMB_OF_ELITE_SCHEDULES, population.getSchedules().size()).forEach(x -> {
            if(Driver.CROSSOVER_RATE > Math.random()) {
                Schedule schedule1 = selectTournamentPopulation(population).sortbyFitness().getSchedules().get(0);
                Schedule schedule2 = selectTournamentPopulation(population).sortbyFitness().getSchedules().get(0);
                crossoverPopulation.getSchedules().set(x, crossoverSchedule(schedule1, schedule2));
            }
            else {
                crossoverPopulation.getSchedules().set(x,  population.getSchedules().get(x));
            }
        });
        return crossoverPopulation;
    }
    public Schedule crossoverSchedule(Schedule schedule1, Schedule schedule2) {
        Schedule crossoverSchedule = new Schedule(data).initialize();
        IntStream.range(0,  crossoverSchedule.getClasses().size()).forEach(x -> {
            if(Math.random() > 0.5)
                crossoverSchedule.getClasses().set(x,  schedule1.getClasses().get(x));
            else
                crossoverSchedule.getClasses().set(x,  schedule2.getClasses().get(x));
        });
        return crossoverSchedule;
    }
```

Рис.3.11 Реалізація зхрещування в класі *GeneticAlgorithm*

Після схрещування відбувається мутація видів також із заданою вірогідністю, а після схрещування йде тест хромосом. (рис. 3.12) Потім сортуємо види за значенням їх придатності.

```
public Population mutatePopulation(Population population) {
    Population mutatePopulation = new Population(population.getSchedules().size(), data);
    ArrayList<Schedule> schedules = mutatePopulation.getSchedules();
    IntStream.range(0, Driver.NUMB_OF_ELITE_SCHEDULES).forEach(x -> schedules.set(x, population.getSchedules().get(x)));
    IntStream.range(Driver.NUMB_OF_ELITE_SCHEDULES, population.getSchedules().size()).forEach(x -> {
        schedules.set(x,  mutateSchedule(population.getSchedules().get(x)));
    });
    return mutatePopulation;
}
public Schedule mutateSchedule(Schedule mutateSchedule) {
    Schedule schedule = new Schedule(data).initialize();
    IntStream.range(0,  mutateSchedule.getClasses().size()).forEach(x -> {
        if(Driver.MUTATION_RATE > Math.random())
            mutateSchedule.getClasses().set(x,  schedule.getClasses().get(x));
    });
    return mutateSchedule;
}
public Population selectTournamentPopulation(Population population) {
    Population tournamentPopulation = new Population(Driver.TOURNAMENT_SELECTION_SIZE, data);
    IntStream.range(0,  Driver.TOURNAMENT_SELECTION_SIZE).forEach(x -> {
        tournamentPopulation.getSchedules().set(x, population.getSchedules().get((int)(Math.random() * population.getSchedules().
    });
    return tournamentPopulation;
```

Рис.3.12 Реалізація зхрещування в класі *GeneticAlgorithm*

Значення придатності це значення що оприділяє приданість даного розкладу до виконання заданих умов. Щоб отримати дане значення був створений клас *Schedule*, який проводить перевірку отриманої популяції на придатність і видає результат в проміжку (0;1].

```
private double calculateFitness() {
    numbOfConflicts = 0;
    classes.forEach(x -> {
        if(x.getRoom().getSeatingCapacity() < x.getCourse().getMaxNumbOfStudents())
            numbOfConflicts++;
        classes.stream().filter(y -> classes.indexOf(y) >= classes.indexOf(x)).forEach(y -> {
            if(x.getMeetingTime() == y.getMeetingTime() && x.getId() != y.getId()) {
                if(x.getRoom() == y.getRoom())
                    numbOfConflicts++;
                if(x.getInstructor() == y.getInstructor())
                    numbOfConflicts++;
            }
        });
    });
    return 1/(double)(numbOfConflicts+1);
}
```

Рис.3.13 Реалізація обрахунку придатності в класі *Schedule*

Останнім класом залишився клас *Driver*, який дозволяє проводити настройку роботи генетичного алгоритму та проводить запуск програми та вивід результатів.

```
public class Driver {
    public static final int POPULATION_SIZE = 20;
    public static final double MUTATION_RATE = 0.3;
    public static final double CROSSOVER_RATE = 0.8;
    public static final int TOURNAMENT_SELECTION_SIZE = 6;
    public static final int NUMB_OF_ELITE_SCHEDULES = 2;
    private int scheduleNumb = 0;
    private int classNumb = 1;
    private Data data;

    private void printScheduleAsTable(Schedule schedule, int generation) {
        ArrayList<Class> classes = schedule.getClasses();
        System.out.print("\n                    ");
        System.out.println("Class # | Dept | Course (number, max # of students) | Room (Capacity)
        System.out.print("                    ");
        System.out.print("-------------------------------------------------");
        System.out.println("-------------------------------------------------------------");
        classes.forEach(x -> {
            int majorIndex = data.getDepts().indexOf(x.getDept());
            int courseIndex = data.getCourses().indexOf(x.getCourse());
            int roomsIndex = data.getRooms().indexOf(x.getRoom());
            int instructorIndex = data.getInstructors().indexOf(x.getInstructor());
            int meetingTimeIndex = data.getMeetingTimes().indexOf(x.getMeetingTime());
            System.out.print("                    ");
            System.out.print(String.format("  %1$02d  ", classNumb) + " | ");
```

Рис.3.13 Реалізація класу *Driver*

Основні з властивостей, які треба налаштувати перед початком роботи алгоритму:

1. *POPULATION_SIZE* - максимальне число видів, яке може містити популяція. Це той розмір, до якого "урізається" популяція після схрещування. За замовчуванням 20.

2. *CROSSOVER_RATE* - ймовірність схрещування. Вона повинна бути в інтервалі (0,0; 1.0]. Якщо ця умова не виконується, то випадає виняток *ArgumentOutOfRangeException*. За замовчуванням це значення встановлено в 0.8.

3. *MUTATION_RATE* - ймовірність мутації. Вона повинна бути в інтервалі [0,0; 1.0). Якщо ця умова не виконується, то випадає виняток *ArgumentOutOfRangeException*.

Отримана в результаті розробки ситема складвється з дванадцяти взаємопов’язаних класів, чиї звязки можна побачити наочно на рисунку 3.14.

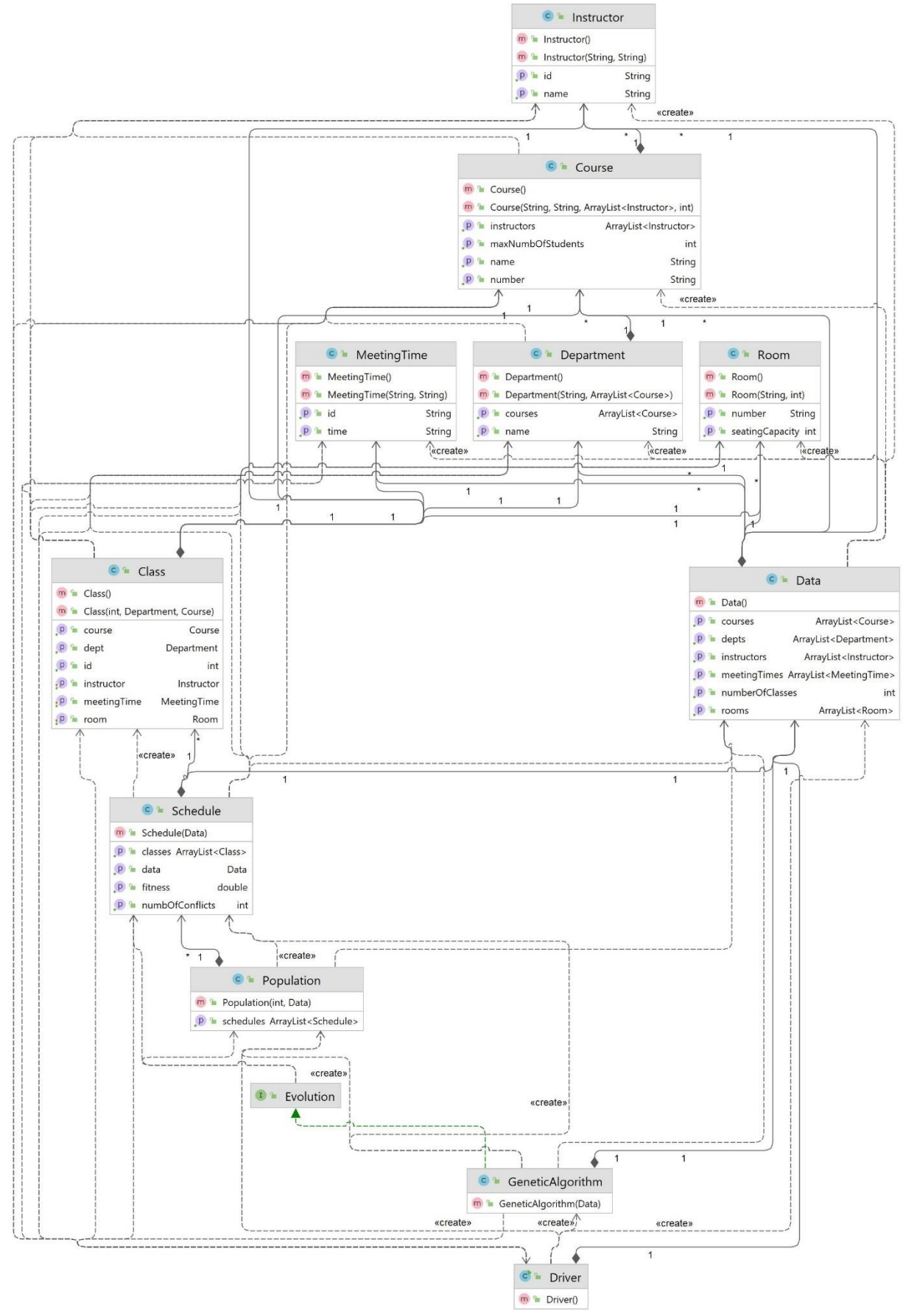


Рис. 3.14 Діаграма класів

3.3. Демонстрація системи

Для демонстрації роботи системи мною було створене тестове завдання з даними наведеними на рисунку 3.15.

```
Наявні кафедри ==>
name: MATH, courses: [15MAT201, 15MAT303]
name: EEE, courses: [15EEE101, 15EEE233, 15EEE400]
name: PHY, courses: [15PHY310, 15PHY222]
Наявні предмети ==>
course #: C1, name: 15MAT201, max # of students: 25, instructors: [Steve Andreson , Kevin Peter    ]
course #: C2, name: 15EEE101, max # of students: 35, instructors: [Steve Andreson , Kevin Peter    , Mike John      ]
course #: C3, name: 15MAT303, max # of students: 25, instructors: [Steve Andreson , Kevin Peter    ]
course #: C4, name: 15EEE233, max # of students: 30, instructors: [Mike John      , William        ]
course #: C5, name: 15EEE400, max # of students: 35, instructors: [William        ]
course #: C6, name: 15PHY310, max # of students: 45, instructors: [Steve Andreson , Mike John      ]
course #: C7, name: 15PHY222, max # of students: 45, instructors: [Kevin Peter    , William        ]
Наявні кабвнети ==>
room #: R1, max seating capacity: 25
room #: R2, max seating capacity: 45
room #: R3, max seating capacity: 35
Наявні викладачі ==>
id: I1, name: Steve Andreson
id: I2, name: Kevin Peter
id: I3, name: Mike John
id: I4, name: William
Наявний час проведення занять ==>
id: MT1, Meeting time: MWF 09:00 - 10:00
id: MT2, Meeting time: TWF 10:00 - 11:00
id: MT3, Meeting time: FRI 09:00 - 10:30
id: MT4, Meeting time: MTW 10:30 - 12:00
```

Рис. 3.15 Тестові дані

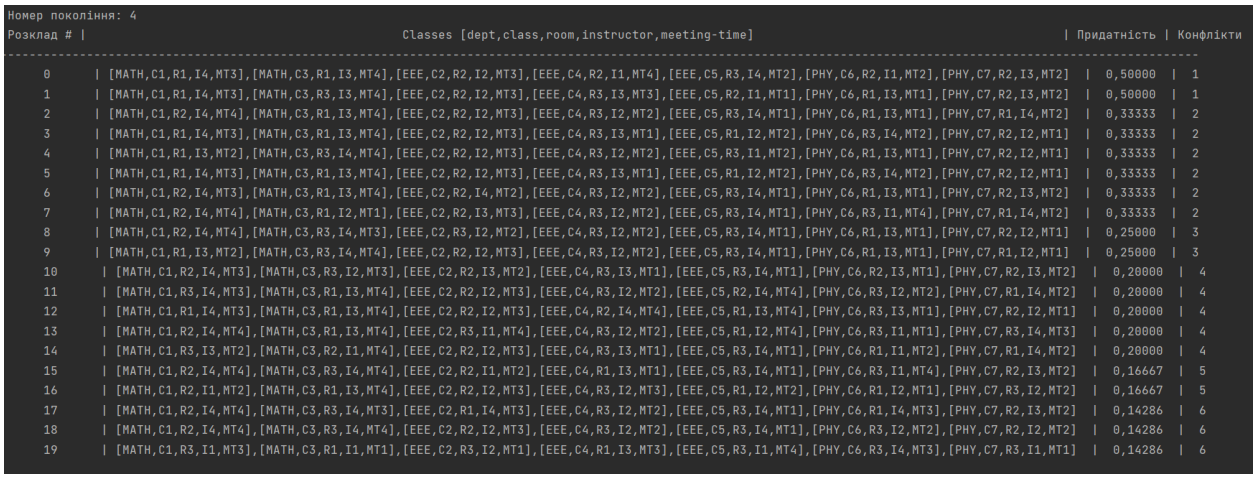
Для початку запустимо систему на стандартних налаштуваннях. Після запуску ми отримаємо нульове покоління – першу популяцію створену нашим алгоритмом апсолютно випадковим способом.(рис. 3.16)

```
> Номер покоління: 0
  Розклад # |                                   Classes [dept,class,room,instructor,meeting-time]                                   | Придатність | Конфлікти
-------------------------------------------------------------------------------------------------------------------------------------------------------
      0     | [MATH,C1,R2,I4,MT4],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R1,I4,MT2]  |  0,33333  |  2
      1     | [MATH,C1,R1,I4,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R3,I1,MT4],[EEE,C4,R2,I1,MT3],[EEE,C5,R1,I2,MT3],[PHY,C6,R1,I2,MT4],[PHY,C7,R2,I3,MT2]  |  0,25000  |  3
      2     | [MATH,C1,R1,I4,MT3],[MATH,C3,R3,I2,MT2],[EEE,C2,R2,I4,MT2],[EEE,C4,R3,I3,MT1],[EEE,C5,R1,I3,MT4],[PHY,C6,R3,I1,MT2],[PHY,C7,R2,I2,MT1]  |  0,25000  |  3
      3     | [MATH,C1,R1,I4,MT2],[MATH,C3,R1,I4,MT4],[EEE,C2,R3,I4,MT1],[EEE,C4,R1,I2,MT4],[EEE,C5,R1,I3,MT3],[PHY,C6,R2,I2,MT3],[PHY,C7,R2,I3,MT2]  |  0,25000  |  3
      4     | [MATH,C1,R1,I4,MT1],[MATH,C3,R2,I3,MT2],[EEE,C2,R2,I1,MT3],[EEE,C4,R2,I4,MT1],[EEE,C5,R1,I4,MT2],[PHY,C6,R2,I2,MT1],[PHY,C7,R3,I2,MT2]  |  0,20000  |  4
      5     | [MATH,C1,R2,I4,MT2],[MATH,C3,R3,I4,MT4],[EEE,C2,R1,I4,MT3],[EEE,C4,R3,I1,MT2],[EEE,C5,R2,I1,MT4],[PHY,C6,R1,I1,MT1],[PHY,C7,R1,I2,MT3]  |  0,20000  |  4
      6     | [MATH,C1,R3,I4,MT2],[MATH,C3,R3,I4,MT2],[EEE,C2,R3,I4,MT3],[EEE,C4,R2,I3,MT1],[EEE,C5,R2,I2,MT1],[PHY,C6,R1,I1,MT1],[PHY,C7,R2,I3,MT4]  |  0,20000  |  4
      7     | [MATH,C1,R2,I2,MT1],[MATH,C3,R1,I3,MT1],[EEE,C2,R3,I4,MT4],[EEE,C4,R3,I4,MT2],[EEE,C5,R3,I4,MT2],[PHY,C6,R3,I2,MT1],[PHY,C7,R2,I3,MT4]  |  0,20000  |  4
      8     | [MATH,C1,R1,I1,MT1],[MATH,C3,R1,I1,MT2],[EEE,C2,R2,I2,MT2],[EEE,C4,R1,I3,MT2],[EEE,C5,R2,I4,MT1],[PHY,C6,R1,I2,MT3],[PHY,C7,R2,I2,MT1]  |  0,20000  |  4
      9     | [MATH,C1,R1,I4,MT4],[MATH,C3,R2,I4,MT3],[EEE,C2,R1,I2,MT4],[EEE,C4,R2,I3,MT3],[EEE,C5,R3,I2,MT4],[PHY,C6,R1,I2,MT1],[PHY,C7,R2,I4,MT1]  |  0,16667  |  5
      10    | [MATH,C1,R2,I1,MT3],[MATH,C3,R3,I3,MT2],[EEE,C2,R3,I4,MT2],[EEE,C4,R1,I4,MT3],[EEE,C5,R1,I3,MT1],[PHY,C6,R2,I1,MT1],[PHY,C7,R3,I3,MT1]  |  0,16667  |  5
      11    | [MATH,C1,R3,I4,MT2],[MATH,C3,R3,I1,MT4],[EEE,C2,R1,I3,MT4],[EEE,C4,R2,I4,MT3],[EEE,C5,R1,I1,MT2],[PHY,C6,R3,I2,MT4],[PHY,C7,R1,I2,MT2]  |  0,14286  |  6
      12    | [MATH,C1,R1,I4,MT2],[MATH,C3,R3,I4,MT4],[EEE,C2,R3,I3,MT4],[EEE,C4,R2,I1,MT2],[EEE,C5,R2,I1,MT2],[PHY,C6,R3,I2,MT4],[PHY,C7,R2,I1,MT3]  |  0,14286  |  6
      13    | [MATH,C1,R3,I2,MT2],[MATH,C3,R2,I1,MT3],[EEE,C2,R3,I3,MT2],[EEE,C4,R2,I4,MT3],[EEE,C5,R2,I2,MT3],[PHY,C6,R1,I3,MT4],[PHY,C7,R1,I3,MT1]  |  0,14286  |  6
      14    | [MATH,C1,R2,I2,MT3],[MATH,C3,R1,I1,MT3],[EEE,C2,R1,I3,MT2],[EEE,C4,R3,I2,MT4],[EEE,C5,R1,I2,MT4],[PHY,C6,R2,I1,MT1],[PHY,C7,R1,I2,MT3]  |  0,14286  |  6
      15    | [MATH,C1,R3,I4,MT3],[MATH,C3,R2,I2,MT2],[EEE,C2,R3,I2,MT2],[EEE,C4,R1,I2,MT4],[EEE,C5,R3,I4,MT4],[PHY,C6,R1,I3,MT4],[PHY,C7,R3,I1,MT2]  |  0,14286  |  6
      16    | [MATH,C1,R2,I1,MT2],[MATH,C3,R1,I1,MT4],[EEE,C2,R2,I2,MT2],[EEE,C4,R2,I3,MT3],[EEE,C5,R1,I2,MT4],[PHY,C6,R3,I3,MT3],[PHY,C7,R1,I2,MT2]  |  0,12500  |  7
      17    | [MATH,C1,R1,I4,MT1],[MATH,C3,R1,I4,MT2],[EEE,C2,R3,I2,MT2],[EEE,C4,R1,I2,MT2],[EEE,C5,R1,I4,MT3],[PHY,C6,R3,I4,MT1],[PHY,C7,R3,I2,MT4]  |  0,12500  |  7
      18    | [MATH,C1,R3,I4,MT1],[MATH,C3,R2,I4,MT4],[EEE,C2,R2,I2,MT4],[EEE,C4,R1,I1,MT4],[EEE,C5,R1,I3,MT4],[PHY,C6,R2,I1,MT4],[PHY,C7,R1,I3,MT1]  |  0,11111  |  8
      19    | [MATH,C1,R3,I2,MT3],[MATH,C3,R3,I2,MT1],[EEE,C2,R1,I2,MT1],[EEE,C4,R1,I1,MT1],[EEE,C5,R2,I2,MT1],[PHY,C6,R2,I3,MT1],[PHY,C7,R1,I2,MT2]  |  0,11111  |  8
```

Рис. 3.16 Нульове покоління

Зрозуміло що нульове покоління не може одразу дати вірний розвязок, адже во створюється випадковим чином, і хоча існує вірогідність такої ситуації, її шанс настільки низький що ним можна знехтувати.

Як видно з рисунка максимальна придатність складає 0,333, що є несприйнятним варіантом. Дана цифра не змінювалась до четвертого циклу де змогла піднятись до 0,5.(рис. 3.17)

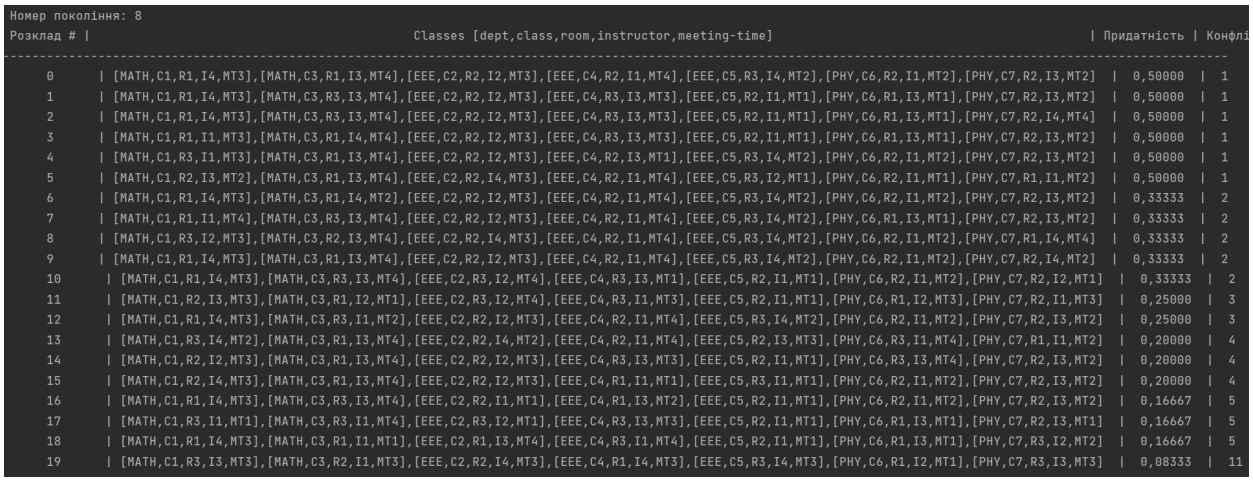


Номер покоління: 4

Розклад #	Classes [dept,class,room,instructor,meeting-time]	Придатність	Конфлікти
0	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,50000	1
1	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,50000	1
2	[MATH,C1,R2,I4,MT4],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R1,I4,MT2]	0,33333	2
3	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT1],[EEE,C5,R1,I2,MT2],[PHY,C6,R3,I4,MT2],[PHY,C7,R2,I2,MT1]	0,33333	2
4	[MATH,C1,R1,I3,MT2],[MATH,C3,R3,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I1,MT2],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I2,MT1]	0,33333	2
5	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT1],[EEE,C5,R1,I2,MT2],[PHY,C6,R3,I4,MT2],[PHY,C7,R2,I2,MT1]	0,33333	2
6	[MATH,C1,R2,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I4,MT2],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,33333	2
7	[MATH,C1,R2,I4,MT4],[MATH,C3,R1,I2,MT1],[EEE,C2,R2,I3,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R3,I1,MT4],[PHY,C7,R1,I4,MT2]	0,33333	2
8	[MATH,C1,R2,I4,MT4],[MATH,C3,R3,I4,MT3],[EEE,C2,R3,I2,MT2],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I2,MT1]	0,25000	3
9	[MATH,C1,R1,I3,MT2],[MATH,C3,R3,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R1,I2,MT1]	0,25000	3
10	[MATH,C1,R2,I4,MT3],[MATH,C3,R3,I2,MT3],[EEE,C2,R2,I3,MT2],[EEE,C4,R3,I3,MT1],[EEE,C5,R3,I4,MT1],[PHY,C6,R2,I3,MT1],[PHY,C7,R2,I3,MT2]	0,20000	4
11	[MATH,C1,R3,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R2,I4,MT4],[PHY,C6,R3,I2,MT2],[PHY,C7,R1,I4,MT2]	0,20000	4
12	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I4,MT4],[EEE,C5,R1,I3,MT4],[PHY,C6,R3,I3,MT1],[PHY,C7,R2,I2,MT1]	0,20000	4
13	[MATH,C1,R2,I4,MT4],[MATH,C3,R1,I3,MT4],[EEE,C2,R3,I1,MT4],[EEE,C4,R3,I2,MT2],[EEE,C5,R1,I2,MT4],[PHY,C6,R3,I1,MT1],[PHY,C7,R3,I4,MT3]	0,20000	4
14	[MATH,C1,R3,I3,MT2],[MATH,C3,R2,I1,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT1],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I1,MT2],[PHY,C7,R1,I4,MT2]	0,20000	4
15	[MATH,C1,R2,I4,MT4],[MATH,C3,R3,I4,MT4],[EEE,C2,R2,I1,MT2],[EEE,C4,R1,I3,MT1],[EEE,C5,R3,I4,MT1],[PHY,C6,R3,I1,MT4],[PHY,C7,R2,I3,MT2]	0,16667	5
16	[MATH,C1,R2,I1,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I2,MT3],[EEE,C5,R1,I2,MT2],[PHY,C6,R1,I2,MT1],[PHY,C7,R3,I2,MT2]	0,16667	5
17	[MATH,C1,R2,I4,MT4],[MATH,C3,R3,I4,MT3],[EEE,C2,R1,I4,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I4,MT3],[PHY,C7,R2,I3,MT2]	0,14286	6
18	[MATH,C1,R2,I4,MT4],[MATH,C3,R3,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I2,MT2],[EEE,C5,R3,I4,MT1],[PHY,C6,R3,I2,MT2],[PHY,C7,R2,I2,MT2]	0,14286	6
19	[MATH,C1,R3,I1,MT3],[MATH,C3,R1,I1,MT1],[EEE,C2,R3,I2,MT1],[EEE,C4,R1,I3,MT3],[EEE,C5,R3,I1,MT4],[PHY,C6,R3,I4,MT3],[PHY,C7,R3,I1,MT1]	0,14286	6

Рис. 3.17 Четверте покоління

Після четвертого покоління з новими поколіннями збільшувалась лише кількість особин з придатністю 0,5. Так на 8 покоління кількість таких особин складала шість представників популяції.(рис. 3.18)



Номер покоління: 8

Розклад #	Classes [dept,class,room,instructor,meeting-time]	Придатність	Конфлі
0	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,50000	1
1	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,50000	1
2	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I4,MT4]	0,50000	1
3	[MATH,C1,R1,I1,MT3],[MATH,C3,R1,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,50000	1
4	[MATH,C1,R3,I1,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I3,MT1],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,50000	1
5	[MATH,C1,R2,I3,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I4,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I2,MT1],[PHY,C6,R2,I1,MT1],[PHY,C7,R1,I1,MT2]	0,50000	1
6	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I4,MT2],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,33333	2
7	[MATH,C1,R1,I1,MT4],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,33333	2
8	[MATH,C1,R3,I2,MT3],[MATH,C3,R2,I3,MT4],[EEE,C2,R2,I4,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R1,I4,MT4]	0,33333	2
9	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I4,MT2]	0,33333	2
10	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R3,I2,MT4],[EEE,C4,R3,I3,MT1],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I2,MT1]	0,33333	2
11	[MATH,C1,R2,I3,MT3],[MATH,C3,R1,I2,MT1],[EEE,C2,R3,I2,MT4],[EEE,C4,R3,I1,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I2,MT3],[PHY,C7,R2,I1,MT3]	0,25000	3
12	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I1,MT2],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,25000	3
13	[MATH,C1,R3,I4,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I4,MT2],[EEE,C4,R2,I1,MT4],[EEE,C5,R2,I3,MT3],[PHY,C6,R3,I1,MT4],[PHY,C7,R1,I1,MT2]	0,20000	4
14	[MATH,C1,R2,I2,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R3,I3,MT4],[PHY,C7,R2,I3,MT2]	0,20000	4
15	[MATH,C1,R2,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R1,I1,MT1],[EEE,C5,R3,I1,MT1],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,20000	4
16	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I1,MT1],[EEE,C4,R1,I3,MT2],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,16667	5
17	[MATH,C1,R3,I1,MT1],[MATH,C3,R3,I1,MT4],[EEE,C2,R3,I2,MT1],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT1]	0,16667	5
18	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I1,MT1],[EEE,C2,R1,I3,MT4],[EEE,C4,R3,I1,MT4],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R3,I2,MT2]	0,16667	5
19	[MATH,C1,R3,I3,MT3],[MATH,C3,R2,I1,MT3],[EEE,C2,R2,I4,MT3],[EEE,C4,R1,I4,MT3],[EEE,C5,R3,I4,MT3],[PHY,C6,R1,I2,MT1],[PHY,C7,R3,I3,MT3]	0,08333	11

Рис. 3.18 Восьме покоління

І хоча на десяте покоління кількість осбин з придатністю 0,5 впала до трьох(рис. 3.19), уже на одинацятому поколінні був знайдений вірний розвязок.(рис. 3.20)

Номер покоління: 10

Розклад #	Classes [dept,class,room,instructor,meeting-time]	Придатність	Конфлікти
0	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,50000	1
1	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,50000	1
2	[MATH,C1,R1,I1,MT3],[MATH,C3,R1,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I2,MT4]	0,50000	1
3	[MATH,C1,R3,I1,MT4],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R1,I1,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I4,MT1],[PHY,C7,R2,I2,MT4]	0,33333	2
4	[MATH,C1,R2,I1,MT2],[MATH,C3,R3,I3,MT4],[EEE,C2,R1,I4,MT2],[EEE,C4,R3,I3,MT3],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I3,MT2]	0,25000	3
5	[MATH,C1,R2,I3,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R3,I3,MT2],[EEE,C4,R2,I1,MT4],[EEE,C5,R2,I4,MT2],[PHY,C6,R2,I1,MT1],[PHY,C7,R1,I1,MT2]	0,25000	3
6	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I2,MT3],[EEE,C2,R1,I2,MT4],[EEE,C4,R2,I2,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,25000	3
7	[MATH,C1,R3,I1,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R3,I1,MT2],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,25000	3
8	[MATH,C1,R1,I2,MT2],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I4,MT3],[EEE,C4,R3,I3,MT1],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I1,MT2],[PHY,C7,R1,I1,MT2]	0,25000	3
9	[MATH,C1,R3,I2,MT3],[MATH,C3,R3,I1,MT4],[EEE,C2,R2,I4,MT3],[EEE,C4,R1,I2,MT3],[EEE,C5,R3,I4,MT2],[PHY,C6,R3,I1,MT3],[PHY,C7,R2,I3,MT2]	0,20000	4
10	[MATH,C1,R3,I2,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I4,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R1,I3,MT4],[PHY,C6,R2,I1,MT1],[PHY,C7,R1,I1,MT2]	0,20000	4
11	[MATH,C1,R2,I3,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I3,MT4],[EEE,C4,R2,I1,MT4],[EEE,C5,R1,I2,MT4],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I1,MT2]	0,16667	5
12	[MATH,C1,R2,I4,MT2],[MATH,C3,R1,I4,MT2],[EEE,C2,R3,I3,MT4],[EEE,C4,R2,I1,MT4],[EEE,C5,R2,I2,MT2],[PHY,C6,R1,I1,MT2],[PHY,C7,R3,I2,MT1]	0,16667	5
13	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R1,I2,MT2],[EEE,C4,R2,I1,MT4],[EEE,C5,R2,I3,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,16667	5
14	[MATH,C1,R1,I1,MT3],[MATH,C3,R1,I2,MT1],[EEE,C2,R2,I4,MT3],[EEE,C4,R1,I3,MT1],[EEE,C5,R2,I3,MT1],[PHY,C6,R2,I1,MT1],[PHY,C7,R3,I4,MT2]	0,16667	5
15	[MATH,C1,R2,I3,MT2],[MATH,C3,R3,I2,MT2],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,16667	5
16	[MATH,C1,R3,I2,MT1],[MATH,C3,R1,I1,MT4],[EEE,C2,R3,I1,MT4],[EEE,C4,R2,I3,MT3],[EEE,C5,R1,I2,MT4],[PHY,C6,R1,I2,MT3],[PHY,C7,R1,I3,MT1]	0,16667	5
17	[MATH,C1,R1,I3,MT3],[MATH,C3,R2,I3,MT3],[EEE,C2,R2,I4,MT3],[EEE,C4,R2,I3,MT1],[EEE,C5,R3,I4,MT1],[PHY,C6,R1,I1,MT2],[PHY,C7,R1,I1,MT2]	0,14286	6
18	[MATH,C1,R3,I2,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R1,I2,MT3],[EEE,C5,R3,I3,MT1],[PHY,C6,R3,I4,MT3],[PHY,C7,R1,I1,MT2]	0,12500	7
19	[MATH,C1,R1,I4,MT3],[MATH,C3,R2,I2,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R1,I4,MT3],[PHY,C6,R3,I4,MT3],[PHY,C7,R2,I4,MT4]	0,11111	8

Рис. 3.19 Десяте покоління

Номер покоління: 11

Розклад #	Classes [dept,class,room,instructor,meeting-time]	Придатність	Конфлікти
0	[MATH,C1,R1,I1,MT3],[MATH,C3,R3,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I3,MT2]	1,00000	0
1	[MATH,C1,R1,I4,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I3,MT2]	0,50000	1
2	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,50000	1
3	[MATH,C1,R1,I1,MT3],[MATH,C3,R1,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R3,I4,MT2],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I3,MT2]	0,50000	1
4	[MATH,C1,R1,I1,MT3],[MATH,C3,R1,I4,MT4],[EEE,C2,R2,I2,MT1],[EEE,C4,R2,I2,MT4],[EEE,C5,R3,I1,MT4],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I3,MT2]	0,50000	1
5	[MATH,C1,R3,I3,MT1],[MATH,C3,R3,I3,MT4],[EEE,C2,R3,I1,MT3],[EEE,C4,R1,I4,MT1],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I3,MT3],[PHY,C7,R1,I1,MT2]	0,33333	2
6	[MATH,C1,R1,I4,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R3,I3,MT1],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R1,I3,MT1],[PHY,C7,R2,I2,MT3]	0,33333	2
7	[MATH,C1,R3,I4,MT4],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R3,I1,MT2],[PHY,C7,R2,I3,MT2]	0,33333	2
8	[MATH,C1,R3,I1,MT4],[MATH,C3,R1,I4,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT1],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I2,MT4]	0,33333	2
9	[MATH,C1,R1,I1,MT3],[MATH,C3,R3,I1,MT2],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT1],[PHY,C7,R1,I2,MT2]	0,33333	2
10	[MATH,C1,R3,I1,MT4],[MATH,C3,R1,I3,MT4],[EEE,C2,R1,I3,MT1],[EEE,C4,R1,I1,MT3],[EEE,C5,R3,I4,MT1],[PHY,C6,R2,I1,MT2],[PHY,C7,R2,I2,MT3]	0,33333	2
11	[MATH,C1,R1,I1,MT3],[MATH,C3,R3,I2,MT3],[EEE,C2,R3,I3,MT3],[EEE,C4,R3,I1,MT1],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I1,MT4],[PHY,C7,R2,I3,MT2]	0,33333	2
12	[MATH,C1,R3,I1,MT4],[MATH,C3,R2,I1,MT2],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT2],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I4,MT3]	0,33333	2
13	[MATH,C1,R2,I3,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R2,I1,MT1],[PHY,C6,R2,I1,MT1],[PHY,C7,R1,I1,MT2]	0,25000	3
14	[MATH,C1,R3,I2,MT3],[MATH,C3,R2,I2,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I4,MT4],[EEE,C5,R3,I3,MT1],[PHY,C6,R2,I4,MT3],[PHY,C7,R1,I1,MT2]	0,25000	3
15	[MATH,C1,R2,I1,MT3],[MATH,C3,R3,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R3,I3,MT3],[EEE,C5,R2,I1,MT1],[PHY,C6,R1,I1,MT1],[PHY,C7,R3,I4,MT1]	0,20000	4
16	[MATH,C1,R2,I2,MT2],[MATH,C3,R3,I2,MT3],[EEE,C2,R2,I4,MT1],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I2,MT3],[PHY,C6,R2,I1,MT1],[PHY,C7,R2,I2,MT4]	0,20000	4
17	[MATH,C1,R2,I3,MT2],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I4,MT4],[EEE,C4,R2,I1,MT4],[EEE,C5,R3,I4,MT3],[PHY,C6,R2,I1,MT1],[PHY,C7,R3,I4,MT3]	0,20000	4
18	[MATH,C1,R3,I2,MT1],[MATH,C3,R1,I1,MT4],[EEE,C2,R3,I1,MT4],[EEE,C4,R2,I3,MT3],[EEE,C5,R1,I2,MT4],[PHY,C6,R1,I2,MT3],[PHY,C7,R1,I4,MT1]	0,16667	5
19	[MATH,C1,R3,I1,MT3],[MATH,C3,R1,I3,MT4],[EEE,C2,R2,I2,MT3],[EEE,C4,R2,I1,MT4],[EEE,C5,R1,I3,MT4],[PHY,C6,R3,I4,MT1],[PHY,C7,R3,I3,MT4]	0,12500	7

Рис. 3.20 Одинацяте покоління

Як показано на рисунку одинацяте покоління має представника з придатністю 1 що означає вирішення задачі і кінець роботи алгоритму. В кінці роботи ми отримаємо готовий розклад, який задовільняє всі поставлені умови.(рис. 3.21)

Class #	Dept	Course (number, max # of students)	Room (Capacity)	Instructor (Id)	Meeting Time (Id)
01	MATH	15MAT201 (C1, 25)	R1 (25)	Steve Andreson (I1)	FRI 09:00 - 10:30 (MT3)
02	MATH	15MAT303 (C3, 25)	R3 (35)	William (I4)	MTW 10:30 - 12:00 (MT4)
03	EEE	15EEE101 (C2, 35)	R2 (45)	Kevin Peter (I2)	FRI 09:00 - 10:30 (MT3)
04	EEE	15EEE233 (C4, 30)	R3 (35)	Mike John (I3)	FRI 09:00 - 10:30 (MT3)
05	EEE	15EEE400 (C5, 35)	R3 (35)	William (I4)	TWF 10:00 - 11:00 (MT2)
06	PHY	15PHY310 (C6, 45)	R2 (45)	Steve Andreson (I1)	MWF 09:00 - 10:00 (MT1)
07	PHY	15PHY222 (C7, 45)	R2 (45)	Mike John (I3)	TWF 10:00 - 11:00 (MT2)

Рис. 3.20 Одинацяте покоління

ВИСНОВКИ ДО РОЗДІЛУ 3

В даному розділі була описанна реалізація генетичного алгоритму з метою вирішення задачі дискретної оптимізації. Були описані принципи реалізації генетичного алгоритму.

Всі елементи були детально та помодульно викладені у розділі вище, описаний процес створення класів, створення інтерфейсу та методів для генетичних алгоритмів на мові програмування *Java*.

Зауважимо що у процесі розробки та тестування були дотримані усі необхідні принципи реалізації алгоритму та було створено систему виконує всі покладені на неї функції.

У кінці розділу булапродемонстрована повністю готова система, що може з мінімальними змінамивлаштована в снуючу систему і виконувати закладені в неї функції.

ВИСНОВКИ

Виконуючи завдання кваліфікаційного проекту, а саме: досліджуючи реалізацію задачі дискретної оптимізації з використанням еволюційного алгоритму, ми ознайомилися з принципами дискретної оптимізації.

У першому розділі детально розглядається питання задачі дискретної оптимізації та методи вирішення таких задач. Так, поняття задачі дискретної оптимізаціїдля було пояснено як задача оптимізації що має дискретну множину розвязків. Задача оптимізації - це набір певної кількості окремих задач оптимізація. В одній задачі надаються вхідні дані та достатньо інформації для досягнення рішення, тоді як проблема оптимізації — це набір окремих проблем, створених однаковим чином. Дискретність множини розвязків пояснює що дана задача завжди матиме скінченну множину можливих рішень.

Також проаналізовано основну класифікацію алгоритмів дискретної оптимізації, розклянуто недоліки та преваги їх реалізації та використання та зроблено висновок, що хоча точні алогоритми стовідсотково надають найкращий розвязок, їх область використання дуже вузька через дуже великі витрати часу на розвязання однієї задачі.

Під час дослідження було визначено, що неточні алгоритми більш ефективні ніж точні стосовно витраченого часу, але набагато складніші в реалізації. Крім цього, ми не можемо бути впевнені що результат буде абсолютно точний, але цей недолік нівелюється можливістю тонкого настроювання отримуваних результатів, наприклад, в задачі комівояжера отримати не лише найкоротший або найдешевший (при внесенні ціни проїзду) шлях, але і найкоротший шлях, що не дорожчий заданої суми.

У другому розділі дослідження ми визначаємо основні алгоритми оптимізації такі як генетичний, світлячків, бур’янистої оптимізації, косяка риб, інспірований кажанами, зозулин пошук, мавп’ячий пошук. Визначаємо переваги і недоліки кожного алгоритму та особливості їх застосування.

Проводимо порівняльну характеристику алгоитмів та приходимо до висновку що в нашому випадку кращим варіантом буде генетичний алгоритм.

Також в цьому розділі ми розглядаємо об'єктно-орієнтоване програмування загалом і програмування на мові *Java* зокрема, як спосіб реалызації еволюційних алгоритмів.

В третьому розділі проводиться реалізація генетичного алгоритму з метою розвязання часткового випадку однієї з класичних задач оптимізації. В нашому випадку це є задача формування розкладу, що є однією з варіцій задачі оптимізації складу, чи як її інакше називають задачі про збирання рюбзака.

При розробці використовуються основні принципи об'єктно-орієнтованого програмування: інкапсуляція, стеження, поліморфізм. Кінцевий проект був реалізований на мові програмування високого рівня, а саме: *Java*.

Найголовніше: вивчаються способи вирішення однієї з основних проблем у створенні сучасних систем, а саме проблеми оптимізації, адже в наше сьогодення найстабільніший спосіб покращення роботи, і отримання підвищених прибатків є оптимізація всіх можливих ступенів роботи.

Усі помилки, виявлені під час розробки, виправлено. Після завершення процесу розробки система успішно виконує поставлену задачу та видає коректні результати.

Описано принципи роботи системи та складено схему опису алгоритму роботи системи. Всі необхідні дані скомпільовані та вже є частиною готового проекту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) В.М. Александрова, Л.О. Соболенко, System Research & Information Technologies, 2014, № 4
2) Коломійцев, О., Голубничий,Д., Коц, Г., Третяк, В., Євстрат, Д., & Лисиця, А. (2020).Задачі дискретної оптимізації та їх постановка для розміщення засобів захисту врозподіленій системі.Збірник наукових праць ΛΌГОΣ, 36-41.https://doi.org/10.36074/20.11.2020.v5.12
3) Гуляницький Л. Ф., Мулеса О. Ю. Прикладні методи комбінаторної оптимізації : навч. посіб. / Л. Ф. Гуляницький, О. Ю. Мулеса; Рецензенти : О. Ф. Волошин, П. І. С тецюк ; М-во освіти і науки України, Київ. нац. ун-т ім. Т. Шевченка. - К. : Видавн.-поліграф. центр "Київ. ун-т", 2016. – 133 с.
4) Mathros [Електроний ресурс]: [Веб-сайт].- [Електронні дані].-Режим доступу: https://www.mathros.net.ua/rozvjazok-zadachi-cilochyselnogo-programuvannja-metodom-gilok-ta-mezh.html
5) Mathros [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://www.mathros.net.ua/rozvjazok-zadachi-pro-najkorotshyj-shljah-vykorystovujuchy-algorytm-bellmana-forda.html
6) Розв’язання квадратичної задачі про призначення [Електроний ресурс]:[Стаття].- [Електронні дані].-Режим доступу: http://matmod.dstu.dp.ua/article/view/154222
7) Wikipedia [Електроний ресурс]:[Веб-сайт].- [Електронні дані].-Режим доступу: https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%96%D0%BC%D1%96%D1%82%D0%B0%D1%86%D1%96%D1%97_%D0%B2%D1%96%D0%B4%D0%BF%D0%B0%D0%BB%D1%83
8) Yang Xin-She. Firefly Algorithm, Stochastic Test Functions and Design optimization // International Journal of Bio-Inspired Computation. 2010. V. 2. N 2. P. 78—84.
9) Mehrabiana A.R., Lucase C. A novel numerical optimization algorithm inspired from weed colonization // Ecological informatics. 2006. No 1. P. 355—366.

10) Henryk Josiński, Daniel Kostrzewa, Agnieszka Michalczuk, Adam Świtoński, "The Expanded Invasive Weed Optimization Metaheuristic for Solving Continuous and Discrete Optimization Problems", The Scientific World Journal, vol. 2014, Article ID 831691, 14 pages, 2014.

11) Yang X.-S., Deb S. Cuckoo search via Levy flights // Proc. of the world Congress on Nature & Biologically Inspired Computing (NaBIC 2009), December 2009, India. IEEE Publications, USA, pp. 210—214.

12) Oliver C. Ibe. Levy Processes / Oliver C. Ibe // Markov Processes for Stochastic Modeling (Second Edition) / Oliver C. Ibe., 2013. – C. 329–347.

13) Tuba M., Subotic M., Stanarevic N. Modified cuckoo search algorithm for unconstrained optimization problems // Proc. of the 5th European Computing Conference (ECC'I1), Paris, France, April 28—30, 2011. pp. 263—268.

14) Yang X.-S. A New Metaheuristic Bat-Inspired Algorithm, in: Nature Inspired Cooperative Strategies for Optimization (NISCO 2010). Studies in Computational Intelligence. Berlin: Springer, 2010. Vol. 284. P.65-74.

15) Doppler effect [Електронний ресурс] // The Editors of Encyclopaedia Britannica. – 2019. – Режим доступу до ресурсу: https://www.britannica.com/science/Doppler-effect.

16) Mucherino A., Seref O. Monkey Search: A Novel Meta-Heuristic Search for Global Optimization // Data Mining, System Analysis and Optimization in Biomedicine, AIP Conference Proceedings. 2007. P. 162—173.

17) Cavaicanti-J'unior G. M., Bastos-Fitho C. J. A., Lima-Neto F. B., Castro R. M. C.S. A Hybrid Algorithm Based on Fish School Search and Particle Swarm Optimization for Dynamic Problems // Internauional Conference on Swarm Intelligence 2011 (ICSI). V. 2. P. 543—552.

18) Vijini Mallawaarachchi. Introduction to Genetic Algorithms — Including Example Code [Електронний ресурс] / Vijini Mallawaarachchi. – 2017. – Режим доступу до ресурсу: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.

19) Cavaicanti-J’unior G. M., Bastos-Fitho C. J. A., Lima-Neto F. B., Castro R. M. C.S. A Hybrid Algorithm Based on Fish School Search and Particle Swarm Optimization for Dynamic Problems // Internauional Conference on Swarm Intelligence 2011 (ICSI). V. 2. P. 543—552.

20) Електронні документації *java* – Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html