Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп’ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Трифонова Артема Дмитровича*
	(ПІБ)
академічної групи	*122М-21-2*
	(шифр)
спеціальності	*122 Комп’ютерні науки*
	(код і назва спеціальності)
освітньої програми	*«122 Комп’ютерні науки»*
	(назва освітньої програми)
на тему:	*Дослідження ефективності складання розкладу занять університету на основі застосування методів штучного інтелекту*

___________________ *А. Д. Трифонов*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	*Проф. Мещеряков Л.І.*			

Рецензент				

Нормоконтролер	*Проф. Лактіонов І.С.*			

Дніпро
2022

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп’ютерних систем
(повна назва)
_____________ М.О. Алексєєв
(підпис) (прізвище, ініціали)
« » ____________ 20 ____ Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *122 Комп’ютерні науки*
(код і назва спеціальності)

студенту *122м-21-2* *Трифонову Артему Дмитровичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження ефективності складання розкладу занять університету на основі застосування методів штучного інтелекту*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 31.10.2022 р. № 1200-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об’єкт досліджень – процес складання розкладу занять університету.

Предмет досліджень – можливість застосування генетичних алгоритмів для складання розкладу занять університету.

Мета НДР – вивчення застосування методів штучного інтелекту для складення розкладу занять університету.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження, основи складення розкладу занять університету.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень полягає у розробці методики застосування алгоритмів штучного інтелекту до процесу складання розкладу занять університету з метою підвищення його якості.

Практична цінність результатів полягає у впровадженні алгоритмів штучного інтелекту у процес складання розкладу занять університету для покращення організації навчального процесу.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє побачити та оцінити безпосереднє використання розробленої методики. В результаті роботи повинен бути розроблений фрагмент системи складання розкладу занять університету.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз предметної галузі та постановка завдання	01.09.2022-30.09.2022
Опис моделей та методів розв’язання задачі	01.10.2022-31.10.2022
Розробка методики застосування генетичного алгоритму до розв’язання задачі складення розкладу занять університету	01.11.2022-01.12.2022

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки покращенню організації навчального процесу в результаті впровадження алгоритмів штучного інтелекту у процес складання розкладу занять університету.

Завдання видав	____________	*Мещеряков Л.І.*
	(підпис)	(прізвище, ініціали)
Завдання прийняв до виконання	____________	*Трифонов А.Д.*
	(підпис)	(прізвище, ініціали)

Дата видачі завдання: *01.09.2022 р.*

Термін подання кваліфікаційної роботи до ЕК *12.12.2022*

РЕФЕРАТ

Пояснювальна записка: 91 стор., 22 рис., 4 додатка, 68 джерел.

Об'єкт дослідження: процес складання розкладу занять університету.

Предмет дослідження: можливість застосування генетичних алгоритмів для складання розкладу занять університету.

Мета роботи: вивчення застосування методів штучного інтелекту для складення розкладу занять університету.

Методи дослідження. Задля вирішення поставлених задач використовувались математичні методи розв’язання повних перебірних задач з метою пошуку локального мінімуму за умови бюджетування часу розв’язання, та методи машинного навчання, зокрема, механізми генетичних алгоритмів, такі, як схрещування, турнірний відбір та мутації.

Новизна отриманих результатів полягає у розробці методики застосування алгоритмів штучного інтелекту до процесу складання розкладу занять університету з метою підвищення його якості.

Практична цінність результатів полягає у впровадженні алгоритмів штучного інтелекту у процес складання розкладу занять університету для покращення організації навчального процесу.

Область застосування. Розроблена методика може буде застосована як основа для створення системи складання розкладу занять університету.

Значення роботи та висновки. В ході роботи були досліджені методи та засоби розв’язання NP-повних задач з метою пошуку локального мінімуму. Був обраний генетичний алгоритм як найбільш раціональний спосіб побудови розкладу занять університету за наявними даними з реальної організації навчального процесу.

Прогнози щодо розвитку досліджень. Необхідно дослідити ефективність прийнятого рішення протягом довшого проміжку часу, ніж час розробки. Спланувати та провести дослідження результатів впровадження розробленої методики в умовах реального складання розкладу занять університету.

Список ключових слів: алгоритми, машинне навчання, складання розкладу, генетичні алгоритми, математична модель, селекція, природний добір, мутація.

ABSTRACT

Explanatory note: 91 pages, 22 figures, 4 appendices, 68 sources.

Object of research: the process of drawing up a schedule of university classes.

Subject of research: the possibility of using genetic algorithms for drawing up the schedule of university classes.

Purpose of Master's thesis: to study the use of artificial intelligence methods for drawing up the schedule of university classes.

Research methods. To solve the problems, mathematical methods of solving complete sorting problems were used in order to find a local minimum under the condition of budgeting the solution time, and methods of machine learning, in particular, mechanisms of genetic algorithms, such as crossing, tournament selection and mutations.

Originality of obtained results consists in the development of methods of applying artificial intelligence algorithms to the process of drawing up the schedule of university classes in order to improve its quality.

Practical value of the results consists in the implementation of artificial intelligence algorithms in the process of drawing up the schedule of university classes to improve the organization of the educational process.

Scope of application. The developed methodology can be used as a basis for creating a system for drawing up a university class schedule.

The value of the work and conclusions. In the course of the work methods and means of solving NP-complete problems with the aim of finding a local minimum were investigated. The genetic algorithm was chosen as the most rational way of building the schedule of university classes based on the available data from the real organization of the educational process.

Research forecast and development. It is necessary to investigate the effectiveness of the adopted decision during a longer period of time than the time of development. To plan and conduct a study of the results of the implementation of the developed methodology in the conditions of real preparation of the schedule of university classes.

Keyword: algorithms, machine learning, scheduling, genetic algorithms, mathematical model, selection, natural selection, mutation.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ЗВО – заклад вищої освіти

ІАК - інтерактивна активація та конкуренція

УР – університетський розклад

УРК - університетський розклад курсів

ШНМ – штучна нейронна мережа

CP - Critical Path

CPGA - Critical Path Genetic Algorithm

DAG - Directed Acyclic Graph

GA - Genetic Algorithm

HD - Heuristic Duplication

MIP – Mixed Integer Problem

ML – Machine learning

NP – Non-Deterministic Polynomial

PESP – Periodic Event Scheduling Problem

SGA - Standard Genetic Algorithm

TDGA - Task Duplication Genetic Algorithm

ЗМІСТ

ВСТУП.. 9
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 11
1.1 Загальні відомості з предметної галузі 11
1.2. Методи розв'язання задачі 14
1.3 Генетичні алгоритми.. 16
1.4. Нейронні мережі.. 19
1.5. Постановка задачі.. 22
1.6. Висновки 25
РОЗДІЛ 2. МОДЕЛІ ТА МЕТОДИ РОЗВ'ЯЗАННЯ ЗАДАЧІ 26
2.1. Стандартний генетичний алгоритм.. 26
2.2 Генетичний алгоритм критичного шляху.. 26
2.3 Генетичний алгоритм дублювання завдань... 39
2.4 Генетичний алгоритм з обмеженнями 42
2.5 Висновки 50
РОЗДІЛ 3. РОЗРОБКА МЕТОДИКИ ЗАСТОСУВАННЯ ГЕНЕТИЧНОГО АЛГОРИТМУ ДО РОЗВ'ЯЗАННЯ ЗАДАЧІ СКЛАДЕННЯ РОЗКЛАДУ ЗАНЯТЬ УНІВЕРСИТЕТУ.. 52
3.1. Обґрунтування вибору алгоритму.. 52
3.2. Вхідні дані.. 54
3.2.1. Навчальний план 54
3.2.2. Перелік груп... 55
3.2.3. Навчальне навантаження... 55
3.2.4. Аудиторний фонд... 56
3.2.5. Обмеження... 57
3.3. Опис алгоритму 57
3.4. Дослідження алгоритму... 60
3.5. Висновки 66
ВИСНОВКИ... 67
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 69
Додаток А. КОД ПРОГРАМИ 76
Додаток Б. ВІДГУК КЕРІВНИКА 89
Додаток В. РЕЦЕНЗІЯ 90

Додаток Г. ПЕРЕЛІК ФАЙЛІВ НА ДИСКУ 91

ВСТУП

Функціонування навчального закладу супроводжується опрацюванням значного обсягу різнорідної інформації. Серед інструментів організації навчального процесу повсюдно застосовується планування та розподіл академічних годин, навантаження викладачів та зайнятості аудиторного фонду у вигляді структурованих взаємопов'язаних документів – розкладу. Вирішення цих завдань покладено на диспетчерські служби та навчально-допоміжний персонал підрозділів.

Розклад навчальних занять – завершальний етап планування навчального процесу. Ретельно продуманий та грамотно складений розклад навчальних занять позитивно впливає на якість організації всього навчального процесу.

Задача складення оптимального розкладу є добре відомою математичною проблемою, що належить до класу невизначено поліноміальних задач (NP-повні задачі), складність розв’язання у цьому класі задач складає O(e^n). З цього опису випливає, що для якісного та оперативного складання та корегування розкладу потрібна методика, заснована на обґрунтованому алгоритмічному розв’язанні перебірної задачі. Завдяки появі методів машинного навчання у математиків та програмістів з’явився новий інструментарій для розв’язання задач такого роду. Серед можливих рішень одне з чільних місць займають генетичні алгоритми, завдяки їх гнучкості та простоті.

Метою кваліфікаційної роботи є вивчення застосування методів штучного інтелекту для складення розкладу занять університету з метою підвищення його якості.

Кваліфікаційна робота складається з трьох розділів. У першому розділі було розглянуто визначення поняття та характеристику процесу складання розкладу, проаналізовані методи розв’язання подібних задач, проведено порівняльний аналіз програмних інструментів задля виявлення їх слабких та сильних сторін. У другому розділі розглянуто види генетичних алгоритмів та

обґрунтовано вибір способу розв’язання повних перебірних задач з метою пошуку локального мінімуму за умови бюджетування часу розв’язання. Третій розділ містить опис та реалізацію методики застосування генетичного алгоритму до розв’язання задачі складання розкладу університету.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Загальні відомості з предметної галузі

Розклад занять – документ, що регламентує часове та календарне здійснення процесу навчання у закладі вищої освіти (ЗВО). Процес складання розкладу ґрунтується на аналізі великої кількості інформації та потребує значних трудовитрат. Ефективне управління ресурсами є одним із першочергових складних завдань, які стоять перед університетами у процесі надання освітніх послуг. Разом з тим, ефективне управління діяльністю закладу освіти в сучасному динамічному зовнішньому та внутрішньому середовищі неможливе без інформаційної підтримки.

Розклад навчальних занять забезпечує:

- послідовність вивчення дисциплін та їх взаємозв'язок;
- відповідність кількості годин з дисциплін та видів навчальних занять робочому навчальному плану;
- рівномірний розподіл навчального навантаження студентів та викладачів протягом семестру;
- раціональне використання аудиторного фонду.

Складання розкладу здійснюється на основі:

- графіка навчального процесу на навчальний рік;
- робочих навчальних планів кожної спеціальності та спеціалізації;
- розподілу студентів на потоки, групи та підгрупи (семінарські/практичні/мовні);
- закріплення навчальних аудиторій за підрозділами.

Завдання планування розкладу навчальних занять ЗВО є завданням на складання розкладу комбінаторного типу, характерною особливістю якої є дуже велика розмірність та наявність великої кількості обмежень складної форми [1].

Завдання складання розкладу ЗВО належить до загальної теорії розкладів. Її можна класифікувати по типу шуканого рішення як завдання упорядкування, по типу цільової функції – як завдання багатокритеріальної оптимізації. Розділ теорії розкладів, до якого належить завдання складання розкладу університету, називається "Складання часових таблиць" [2]. Завдання складання тимчасових таблиць можна охарактеризувати як проблему розміщення певних ресурсів, з урахуванням детермінованих обмежень, в обмежені часові інтервали та місця з метою задовільнити цілу низку поставлених критеріїв оптимальності у максимально можливій мірі [3]. У разі завдання складання розкладу занять у часові інтервали необхідно розмістити заняття навчальних студентських груп.

У найбільш загальному формулюванні завдання складання розкладу полягає в наступному: за допомогою деякої множини ресурсів або обслуговуючих пристроїв має бути виконана деяка фіксована система завдань. Мета полягає в тому, щоб при заданих властивостях завдань і ресурсів та накладених на них обмеженнях знайти ефективний алгоритм упорядкування завдань, що оптимізують або прагне оптимізувати необхідну міру ефективності.

Загальна теорія розкладів передбачає, що всі обслуговуючі пристрої (чи процесори) не можуть виконувати у даний момент часу більше одного завдання, що не є достатнім для розкладу навчальних занять, якщо у якості процесора під час розподілу завдань прийняти навчальну аудиторію. Так у деяких випадках в одній аудиторії можуть проводитися заняття з більш ніж однією групою одночасно, наприклад, загальні лекції для кількох потоків. Тому при перенесенні загальної теорії розкладів на розклад навчальних занять було зроблено такі припущення:

- у якості безлічі завдань для розподілу виступають навчальні заняття викладача з навчальними групами;
- модель часу у системі є дискретною; весь розподіл передбачається періодично повторюваним протягом деякого часового інтервалу;
- усі завдання виконуються за однаковий час, який приймається за одиницю дискретизації часового інтервалу;

• завдання мають приналежність до об'єктів, у якості яких виступають навчальні групи та викладачі [4].

У задачі складання розкладу виділяють обмеження жорсткі та м’які [5, 6]. Більшість авторів до жорстких вимог відносять вимоги, порушення яких у розкладі призводить до неможливості організації на його основі повноцінного навчального процесу (наприклад, вимога «одночасно один викладач може проводити лише одне заняття»). Інші вимоги відносять до м’яких, їм призначають пріоритети, менші порівняно зі жорсткими вимогами. Крім того, за певних умов допускається їхнє повне або часткове невиконання. Зокрема, в реальних завданнях через суперечності між вимогами нерідко не існує розкладу, що задовольняє всім накладеним на нього м’яким вимогам [2, 7]. Аналізуючи роботи з опису завдання складання розкладу, можна назвати такі групи обмежень:

1. Жорсткі обмеження. Мають абсолютний пріоритет, враховуються завжди і не можуть бути порушені. До них відносяться:

- заняття не можуть проводитися одночасно у двох різних аудиторіях;
- потік може мати лише одне заняття в момент часу;
- викладачеві може бути призначене лише одне заняття в одній аудиторії на момент часу;
- місткість аудиторії має бути не меншою за кількість студентів у потоці;
- технічна оснащеність аудиторії повинна відповідати вимогам заняття, що проводиться в ній;
- кількість занять на тиждень має дорівнювати кількості занять на тиждень у семестровому графіку.

2. М’які обмеження. Можуть бути порушені, але це небажано. Порушення цих обмежень можна оцінити, використовуючи систему штрафів. До м’яких обмежень можна віднести:

- у розкладі потоку не повинно бути вікон;
- у потоку на день має бути не менше двох пар;
- необхідно мінімізувати переходи груп між корпусами та ці переходи мають бути на великій перерві;
- враховувати закріплення аудиторій за кафедрами.

Навчальний процес університету формується з урахуванням блочного навчання з двотижневим циклом чергування тижнів. Виходячи з цього, можна додати такі м’які обмеження: бажано, щоб заняття з дисципліни проходили в одній аудиторії та одночасно протягом усього періоду вивчення. Для оцінки розкладу, як правило, застосовується скалярна функція, що ґрунтується на виваженій системі штрафів, зіставлених вимогам за їхнє повне або часткове невиконання. Вагові коефіцієнти, зіставлені окремим вимогам, зазвичай задаються експертом таким чином, щоб вимогам вищого пріоритету відповідали великі значення вагових коефіцієнтів.

Якщо навчальні плани складено таким чином, що дисципліни можуть проводитися лише один раз на тиждень або через тиждень та суперечливість вимог усунута, то задача складання оптимального розкладу може бути представлена як задача лінійного цілочисельного програмування. За такого підходу інтереси суб'єктів навчального процесу враховуються у вигляді обмежень чи редукованих критеріїв оптимальності. Цільовою функцією задачі є мінімізація суми штрафів за порушення м’яких обмежень, що накладаються на розклад. Система обмежень складається з формалізованих жорстких обмежень, які мають бути виконані обов'язково.

1.2. Методи розв’язання задачі

Планування — це задача виконання набору заданих завдань набором обмежених ресурсів. Серед важливих задач планування, задача університетського розкладу (УР) привертає велику увагу як штучного інтелекту,

так і дослідження операцій [14]. У класичних задачах УР набір подій (таких як курси та іспити) призначається кільком інтервалам «класного часу», щоб задовольнити набір обмежень [13].

Задачі УР можуть полягати в розкладі курсів або іспитів. У розкладі іспитів одна з цілей полягає в тому, щоб розподілити кілька іспитів якомога рівномірніше, тоді як у розкладі курсу студенти вимагають максимально компактного розкладу [15]. У кваліфікаційній роботі розглядається клас задач університетського розкладу, відомий як задача університетського розкладу курсів (УРК).

Проблема УРК є NP-складною [16] і включає два рішення: призначення викладача та розклад занять. У призначенні викладача визначається, які курси представлятиме який викладач. У розкладі занять вказується, в якій аудиторії буде представлений кожен курс.

Рішення повинні прийматися таким чином, щоб відповідати ряду обмежень. Враховується професійна кваліфікація викладача, уподобання щодо курсів і днів, розумний розподіл понаднормової роботи між викладачами та наявність обладнання. Крім того, це дозволяє уникнути будь-яких конфліктів у розкладах викладачів і аудиторій.

Обмеження проблеми УРК можна розділити на дві групи - жорсткі та м'які [17]. Поки всі жорсткі обмеження дотримані та задовільнені, розклад здійсненний. Типовим прикладом жорстких обмежень є те, що жоден викладач може викладати щонайбільше один курс за раз. З іншого боку, м’які обмеження – це ті вимоги, які, незважаючи на те, що вони є істотними, мають бути виконані настільки, наскільки це можливо. Тобто в разі потреби вони можуть бути порушені. Як результат, м’які обмеження можна назвати перевагами та використовувати для оцінки якості рішення. Наприклад, типове м’яке обмеження полягає в розподілі класів якомога рівномірніше. Основна мета полягає в тому, щоб знайти можливий розклад, який задовольняє всі жорсткі обмеження та максимізує задоволення як викладачів, так і аудиторій на основі їхніх уподобань. Це дорівнює мінімізації порушення м’якого обмеження.

Проблеми УРК зазвичай відрізняються від одного університету до іншого [14, 18]. Дуже ймовірно, що кожен університет має власний унікальний набір вимог щодо ефективного використання своїх ресурсів, виконання вимог свого бізнесу, забезпечення високого рівня задоволеності своїх студентів і так далі. Отже, для відповідності всім цим унікальним вимогам необхідно розробити індивідуальну систему планування курсу.

Спочатку описуються різні аспекти проблеми. Математичне формулювання задачі потім представляється у вигляді цілочисельної лінійної програми. Використовуючи вказане програмне забезпечення математичного програмування, розв’язується модель. Останнім часом зростає інтерес до метаевристик для вирішення задач УРК. Такі алгоритми включають пошук табу від Lü і Hao [19], імітований відпал від Zhanget al. [20] та генетичний алгоритм від Wang [21].

Оскільки задача є NP-складною, найефективнішими алгоритмами для її вирішення є метаевристики. Приклади цих алгоритмів включають евристику розфарбовування графів [14], пошук табу [19, 25], імітований відпал [20], еволюційні алгоритми [22], міркування на основі випадків [23], двоетапні евристичні алгоритми [24], колонія мурашок [26] тощо. Для вирішення задачі у великих ЗВО використовуються метаевристики, засновані на алгоритмах штучного імунітету, генетичних та імітованих відпалів. Ці алгоритми використовують нові процедури, такі як оператори переміщення та перетину.

1.3 Генетичні алгоритми

Генетичний алгоритм (Genetic Algorithm, GA) призначений для вирішення деяких галузевих проблем, які важко вирішити звичайними методами. GA були вперше представлені Джоном Холландом в Мічиганському університеті в 1975 році [28]. Сьогодні GA — це добре відомі популяційні

еволюційні алгоритми, які вирішують задачі як дискретної, так і безперервної оптимізації. У своїй загальній формі GA імітує дарвінівську еволюцію. Він складається з проблем кодування, методів відбору та генетичних операторів, таких як кросинговер і мутація. GA зазвичай має чотири етапи: відбір, рекомбінація (кросинговер), мутація та заміна.

Узагальнений GA шукає простір рішень із популяцією хромосом, кожна з яких представляє кодоване рішення. Значення придатності призначається кожній хромосомі відповідно до її характеристик. Чим краще хромосома, тим вищим стає це значення. Популяція розвивається за допомогою набору операторів, доки не буде досягнуто певного критерію зупинки.

Типова ітерація GA, генерація, відбувається наступним чином. Найкращі хромосоми поточної популяції безпосередньо копіюються наступним поколінням (відтворення). Механізм відбору вибирає хромосоми поточної популяції таким чином, щоб надати більший шанс хромосомам з вищим значенням придатності. Функції відбору можуть бути такими, як випадковий вибір, пропорційний вибір фітнесу, рейтинговий вибір або турнірний вибір [27]. Відібрані хромосоми схрещуються для створення нового потомства. Існують різні методи кросинговеру, які можуть бути застосовані, наприклад, кросинговер k-точок, рівномірний кросинговер, частково відображений кросинговер, реберний кросинговер, циклічний кросинговер тощо [27]. Після процесу схрещування кожне потомство може мутувати за допомогою іншого механізму, який називається мутацією. Після цього нова популяція знову оцінюється, і весь процес повторюється. Схема загального генетичного алгоритму [27] наведена на рис. 1.1.

```
begin
  INITIALIZE population p with random candidate solution EVALUATE
  each candidate repeat
    SELECT parents from p
    CROSSOVER pairs of parents to produce offspring
    MUTATE offspring
    EVALUATE the offspring
    REPLACEMENT of an individual in p by selecting individual among
    the current individuals and the new offspring to form the population
    for the next generations
  until TERMINATION CONDITION is satisfied;
```

Рис.1.1. Узагальнена схема GA.

Механізми ініціалізації та відбору GA починається з певної кількості хромосом, кожна з яких представляє можливе рішення. Кількість хромосом — це розмір популяції, позначений *pop*. Початкові хромосоми генеруються випадковим чином із можливих рішень. Після ініціалізації алгоритмів кожна хромосома оцінюється та визначається її придатність (тобто цільова функція). Імовірність відбору хромосоми k для механізму схрещування є наступною [12]:

$$p_k = \frac{fit(k)}{\sum_{h=1}^{pop} fit(h)}, \tag{1.1}$$

де *fit(k)* - придатність хромосоми k.

Нові рішення утворюються шляхом схрещування двох інших батьків за допомогою оператора, що називається crossover. Оператори схрещення повинні уникати генерації нездійсненних рішень. Мета полягає в отриманні кращого потомства. Щоб прийняти краще рішення для популяції, визначається нове рішення, яке успадковує від обох батьків. Насправді це об'єднання двох батьків для створення нового рішення. Після схрещення кожне рішення змінюється оператором мутації. Основною метою застосування мутації є уникнення конвергенції до локального оптимуму та диверсифікація популяції. Після отримання мутованого потомства його порівнюють з існуючими рішеннями в популяції. Якщо потомство краще за особину в популяції, воно замінює цю особину та приєднується до популяції на фазі заміщення. Етапи від вибору до заміни повторюються, доки не будуть задоволені критерії зупинки. У цей момент

цикл виходить і повертається найкраща особина популяції. Цей індивід представляє найкраще рішення, знайдене алгоритмом.

Це дуже простий GA. Цей алгоритм можна додатково вдосконалити, щоб знайти ще краще рішення, запровадивши локальний метод оптимізації перед заміною. Метод локальної оптимізації гарантує, що дитина оптимальна, перш ніж її представити популяції. Також треба зауважити, що на кожній ітерації еволюції замінюється лише одна особина. Цей тип GA називається генетичним алгоритмом стаціонарного стану.

1.4. Нейронні мережі

Штучну нейронну мережу (ШНМ) можна визначити як систему, що складається з одиниць розподіленої паралельної обробки, яка обчислює певні прості математичні функції (зазвичай нелінійні) [32]. Ключовим елементом у структурі ШНМ є нейрон. Ця структура натхненна форматом людського мозку, який складається з величезної мережі клітин, які називаються нейронами [33].

Перша модель штучного нейрона є спрощенням того, що до того часу було відомо про біологічний нейрон, так що їхній математичний опис закінчився моделлю з n точками входу $x_1, x_2, ..., x_n$ (імітація дендрита) і тільки одна точка відправлення y (імітація аксона). Щоб імітувати поведінку синапсів, точки входу мають вагові коефіцієнти $w_1, w_2, ..., w_n$, значення яких є негативними або позитивними залежно від того, чи були відповідні синапси гальмівними чи збуджуючими. Таким чином, ефект певного синапсу i в нейроні після синапсу визначається $x_i w_i$. Вага w_i визначає ступінь, до якої нейрон повинен враховувати ознаки активації, що виникають у цьому зв'язку [34]. Виходячи з цієї структури, складаються стратегічні переваги ШНМ порівняно з іншими підходами до вирішення проблем. Безумовно, найбільш значущим є її висока потужність обробки, яка походить від їх структури та масивного паралельного розподілу, а також здатності навчатися та поширюватися [32, 34].

ШНМ можна класифікувати як архітектуру мережі. По суті, існує три типи архітектури: ациклічні мережі з одним шаром, ациклічні мережі з кількома рівнями та рекурентні мережі [32], [34]. В ациклічних мережах потік даних через мережу завжди відбувається від входу до виходу. Вони можуть бути одношаровими, наприклад, персептрон і адалін, або багатошаровими, наприклад багатошаровий персептрон [34].

Рекурентні мережі відрізняються за своєю архітектурою від інших, представляючи деякі нейрони як вхідні дані для вихідних даних самих себе або інших після них. Іншими словами, вихід будь-якого нейрона і-го шару мережі використовується як вхідні вузли в рівнях з індексом, меншим або рівним і. Заявники на мережу можуть мати або не мати зв’язків із самооцінкою і, крім того, можуть також мати приховані рівні, чи ні [34].

Крім архітектури мережі, існують інші важливі функції, які служать для класифікації ШНМ, серед них можна виділити процес навчання (або тренування) і типи ШНМ. Навчання можна класифікувати як під наглядом (у присутності вчителя) і без нагляду (без вчителя). У першому випадку, надаються шаблони мережі або моделі входу та значення їх відповідних виходів, яким повинна слідувати ШНМ. У неконтрольованому навчанні ШНН навчається без урахування шаблону, пов’язаного з кожним екземпляром навчання [32], [34].

ШНМ все ще є мало вивченим методом вирішення проблеми складання УР. Її застосування до цієї проблеми вперше з’явилося в літературі на переході між 1980-ми та 1990-ми роками [30], [35], [36].

У дослідженні [37] розглядаються три основні моделі, які розглядають складання УРК через ШНМ: 1) Інтерактивна активація та конкуренція; 2) Нейронна мережа Поттса та 3) Модифікована нейронна мережа Хопфілда. Перша модель [37] використовує гібридну форму нейронної мережі, що включає звичайну структуру штучного інтелекту для розгортання завдань для нейронної моделі, описану під назвою «Інтерактивна активація та конкуренція» (ІАК). Структура моделі організовує класи послідовно, в першу чергу з мережі в регіоні з найбільшою кількістю обмежень. У той же час мережа конфігурує паралельну

комбінацію ресурсів, найбільш відповідну для розглянутого класу, за умови одночасної взаємодії складних обмежень. Обмеження передаються через коригування синаптичних ваг перед тим, як наступний клас вибирається для масштабування. Результати, отримані за допомогою моделі, заснованої на IAC, показують, що розмір мережі зростає значно повільніше (лінійно) з розміром проблеми та є гнучким, щоб кодифікувати обмеження більш реалістично [30].

У другій моделі [37], нейронної мережі Поттса, пропозиція є похідною нейронної мережі Хопфілда [31], яка використовує, замість звичайних двох стадій нейрона, мультистан нейрона Поттса, а також факторізацію, що забезпечує суттєве скорочення у кількості нейронів [31].

Таким чином, припускаючи, що *L*-заняття повинні бути розподілені (включаючи попередню комбінацію *T*-викладачів із *C*-класами та дисциплінами) у *P*-періоди часу та в *R*-класах, виявилося, що при використанні нейронів з кількома станами мережа повинна $XL \times P$ нейронів періоду та $YL \times R$ нейронів класу. Попередні результати, отримані за допомогою цієї моделі, свідчать про те, що цей підхід є прийнятним як метод генерації кар’єрних шляхів у порівнянні з результатами ручного методу [31].

Третя і остання досліджувана модель [37] є модифікованою формою мережі Хопфілда [31]. Основна перевага цього інструменту полягає в його потенціалі для швидкої обчислювальної потужності при реалізації в апаратному забезпеченні, а також у паралельній природі ШНМ [29], [33]. Першим кроком для відображення задачі оптимізації в цій моделі є вираження одночасно витрат і обмежень проблеми в термінах штрафу. Потім визначається функція енергетичної мережі, переконавшись, що ваги мережі та елемента зміщення визначені правильно [29].

Зміни, внесені до оригінальної моделі мережі Хопфілда, пов’язані з пошуком балансу між штрафними термінами, пошуком більш точних локальних мінімумів (відразу після деяких ранніх ітерацій, зроблених мережею), стохастичністю в мережі та включенням механізму стійкості, який полягає у припиненні раннього падіння у випадку локальних мінімумів [29].

Результати [37], отримані за допомогою підходу модифікованої нейронної мережі Хопфілда, довели, що він порівнянний з найкращими технічними евристичними методами вирішення проблеми. Таким чином, було показано, що мережа здатна виробляти високоякісні рішення надзвичайно складних проблем розподілу часу. Цей метод також кращий за дві вище описані моделі, оскільки він має справу з проблемами однакового розміру та складності, а також з більшими проблемами [29].

1.5. Постановка задачі

Задача УРК складається з набору з l викладачів, набору c курсів і набору r аудиторій. Дано d днів, щоб запланувати їх, і для кожного дня існують певні періоди часу. У кожен період часу в кожній аудиторії може бути представлений один курс. Кожен курс може бути представлений лише в підмножині аудиторій і днів. Мета полягає в тому, щоб максимізувати вимоги викладачів і мінімізувати кількість використовуваних аудиторій.

Задача УРК зазвичай моделюються формулюванням цілочисельного лінійного програмування. Розроблена математична модель представлена в наступному розділі кваліфікаційної роботи. Позначення та параметри, що використовуються в моделі, наступні:

l - кількість викладачів

c - кількість курсів

r - кількість аудиторій

d - кількість днів

t - індекс робочих днів $\{1, 2, ..., d\}$

i - індекс для викладачів, де $\{1, 2, ..., l\}$

j - індекс для курсів, де $\{1, 2, ..., c\}$

k - індекс аудиторій $\{1, 2, ..., e\}$

h - індекс для періоду часу $\{1, 2, 3\}$

$u^1_{i,j}$- можливість викладача i для викладання курсу j

$u_{i,t}^{2}$- можливість викладача i для викладання курсу t

$u_{j,t}^{3}$- можливість викладача j для викладання курсу t

$g_{i,t}$ - 1, якщо викладача i можна запросити в день t, і 0 в іншому випадку

$s_{i,j}$ - 1, якщо викладач i може викладати курс j, і 0 в іншому випадку

$v_{j,k}$ - 1, якщо курс j можна представити в аудиторії k, і 0 в іншому випадку

Змінні рішення:

$X_{i,j,t,l,h}$ - 1, якщо викладач i викладає курс j в день t у класі l протягом періоду часу h, і 0 в іншому випадку

$Y_{i,t}$ – двійкова змінна, яка приймає значення 1, якщо викладача i запрошено в день t, і 0 в іншому випадку.

Модель має наступний вигляд:

$$Maximize\ Z = \\ = \sum_{i=1}^{l}\sum_{j=1}^{c}\sum_{t=1}^{d}\sum_{k=1}^{e}\sum_{h=1}^{3} X_{i,j,t,k,h} \cdot u_{i,j}^{1} \\ + \sum_{i=1}^{l}\sum_{t=1}^{d} Y_{i,t} \cdot u_{i,t}^{2} + \sum_{i=1}^{l}\sum_{j=1}^{c}\sum_{t=1}^{d}\sum_{k=1}^{e}\sum_{h=1}^{3} X_{i,j,t,k,h} \cdot u_{j,t}^{3} \tag{1.2}$$

На предмет:

$$\sum_{i=1}^{l}\sum_{t=1}^{d}\sum_{k=1}^{e}\sum_{h=1}^{3} X_{i,j,t,k,h} = 1 \quad \forall_{j} \tag{1.3}$$

$$\sum_{j=1}^{c}\sum_{k=1}^{e} X_{i,j,t,k,h} \leq Y_{i,t} \quad \forall_{i,t,h} \tag{1.4}$$

$$Y_{i,t} \leq g_{i,t} \quad \forall_{i,t} \tag{1.5}$$

$$\sum_{i=1}^{l}\sum_{j=1}^{c} X_{i,j,t,k,h} \leq 1 \ \forall_{t,k,h}$$

(1.6)

$$\sum_{t=1}^{d}\sum_{k=1}^{e}\sum_{h=1}^{3} X_{i,j,t,k,h} \leq s_{i,j} \ \forall_{i,j}$$

(1.7)

$$\sum_{i=1}^{l}\sum_{t=1}^{d}\sum_{h=1}^{3} X_{i,j,t,k,h} \leq v_{j,k} \ \forall_{j,k}$$

(1.8)

$$Y_{i,t} \in \{0,1\} \ \forall_{i,t}$$

(1.9)

$$Z_{i,j,t,k,h} \in \{0,1\} \ \forall_{i,j,t,k,h}$$

(1.10)

Рівняння (1.2) обчислює загальну успішність. Набір обмежень (1.3) гарантує, що кожен курс викладається. Набір обмежень (1.4) визначає дні, протягом яких викладач має навчати. Крім того, це гарантує, що кожен викладач представляє щонайбільше один курс у будь-який часовий проміжок. Набір обмежень (1.5) забезпечує запрошення викладачів у ті дні, яким вони віддають перевагу. Набір обмежень (1.6) запобігає перехресному призначенню, тобто щонайбільше один курс може бути представлений у кожній аудиторії одночасно. Набір обмежень (1.7) полягає в тому, щоб переконатися, що кожному викладачу призначено курси занять. Набір обмежень (1.8) визначає, що кожен курс призначається відповідним аудиторіям. Набори обмежень (1.9) і (1.10) визначають двійкові змінні рішення.

Метою кваліфікаційної роботи є вивчення застосування методів штучного інтелекту для складення розкладу занять університету шляхом пошуку, порівняння та поліпшення вже наявних алгоритмів.

Для досягнення поставленої мети необхідно дослідити наявні математичні методи розв’язання повних перебірних задач з метою пошуку локального мінімуму за умови бюджетування часу розв’язання, та методи машинного навчання, зокрема, механізми генетичних алгоритмів.

Після проведених досліджень необхідно проаналізувати зібрані відомості та на підставі результатів аналізу запропонувати власну методику застосування алгоритмів штучного інтелекту до процесу складання розкладу занять університету.

1.6. Висновки

У першому розділі були розглянуті та порівняні між собою засоби розв’язання NP-повних задач з метою пошуку локального мінімуму. Наведений аналіз демонструє, що найчастіше для розв’язання перебірних задач використовуються генетичні алгоритми, як найзручніший інструмент за умови задовільної якості та часу розв’язання задачі.

Було наведено визначення цільової функції для планування розкладу курсів. Був обраний генетичний алгоритм як найбільш раціональний спосіб побудови розкладу занять університету за наявними даними з реальної організації навчального процесу.

РОЗДІЛ 2

МОДЕЛІ ТА МЕТОДИ РОЗВ'ЯЗАННЯ ЗАДАЧІ

2.1. Стандартний генетичний алгоритм

Загалом, алгоритми планування завдань можна розділити на два основні класи; жадібні та нежадібні (ітераційні) алгоритми [42]. Основний принцип ітераційних алгоритмів полягає в тому, що вони відходять від початкового рішення і намагаються його вдосконалити. Жадібні алгоритми планування завдань можна розділити на дві категорії: алгоритми з дублюванням і алгоритми без дублювання. Одним із поширених алгоритмів у першій категорії є евристичний алгоритм планування дублювання (DSH) [39], одним із найкращих алгоритмів у другій категорії є алгоритм Modified Critical Path (MCP) [43].

Генетичні алгоритми широко використовуються для отримання високоякісних рішень або навіть оптимальних рішень для широкого діапазону задач комбінаторної оптимізації, включаючи проблему планування завдань [40, 41]. Ще одна перевага генетичного пошуку полягає в тому, що його властивий паралелізм можна використати для подальшого скорочення часу його виконання. У пошуковій системі GA є дві важливі, але конкуруючі теми; необхідність селективного тиску, щоб GA могла зосередити пошук на перспективних областях пошукового простору, і потреба в різноманітності населення, щоб важлива інформація (конкретні значення бітів) не була втрачена [44, 45].

Останнім часом було розроблено декілька GA для вирішення проблеми планування завдань, головною відмінністю серед яких є хромосомне представлення графіка [46, 47, 48, 49, 50, 51, 40]. У другому розділі кваліфікаційної роботи запропоновано два гібридні генетичні алгоритми, які називаються генетичним алгоритмом критичного шляху (Critical Path Genetic Algorithm, CPGA) і генетичним алгоритмом дублювання завдань (Task Duplication Genetic Algorithm, TDGA). Дані алгоритми демонструють ефект об'єднання жадібних алгоритмів із генетичним. Перший алгоритм CPGA

заснований на тому, як ефективно використовувати ідеальний час процесорів і перепланувати критичні вузли шляху, щоб скоротити час їх запуску. Нарешті, одна за одною застосовано дві фітнес-функції. Перша фітнес-функція пов’язана з тим, як мінімізувати загальний час виконання (тривалість розкладу), а друга — із задоволеністю балансу навантаження. Другий алгоритм TDGA заснований на принципі дублювання завдань для мінімізації накладних витрат на зв'язок.

Модель системи складання УРК, яка розглядається в даній кваліфікаційній роботі, може бути описана наступним чином [41]: Система складається з обмеженої кількості повністю з’єднаних однорідних процесорів. Нехай граф завдань G є спрямованим ациклічним графом (Directed Acyclic Graph, DAG), що складається з N вузлів $n_1, n_2,..., n_N$, кожен вузол називається завданням графа, який, у свою чергу, є набором інструкцій, які повинні виконуватися послідовно без випередження в тому самому процесорі. Вузол має один або більше входів. Коли всі вхідні дані доступні, вузол запускається для виконання. Вузол без батьківського елемента називається вузлом входу, а вузол без дочірнього елемента називається вузлом виходу. Вага називається вартістю обчислення вузла n_i і позначається вагою (n_i). Граф також має E-спрямовані ребра, що представляють частковий порядок серед завдань. Частковий порядок запроваджує DAG з обмеженим пріоритетом і передбачає, що якщо $n_i \rightarrow n_j$, то n_j є дочірнім, який не може початися, доки не завершить його батьківський n_i. Вага ребра називається вартістю зв'язку ребра і позначається $c(n_i, n_j)$. Ці витрати виникають, якщо n_i та n_j заплановано на різних процесорах, і вважаються нульовими, якщо n_i та n_j заплановано на одному процесорі. Якщо вузол n_i заплановано для процесора P, час початку та час завершення вузла позначаються $ST(n_i, p)$ та $FT(n_i, p)$ відповідно. Після запланування всіх вузлів довжина розкладу визначається як $maxFT(n_i, p)$ для всіх процесорів. Мета задачі планування завдань полягає в тому, як знайти призначення та час початку завдань процесорам, щоб довжина розкладу була мінімізована і, в той же час, зберігалися обмеження пріоритету. Критичний шлях (Critical Path, CP) графа завдання

визначається як шлях із максимальною сумою ваг вузлів і ребер від вузла входу до вузла виходу. Вузол у CP позначається вузлами CP (CPN).

Приклад DAG [38] представлено на рис. 2.1, де CP намальовано жирним.

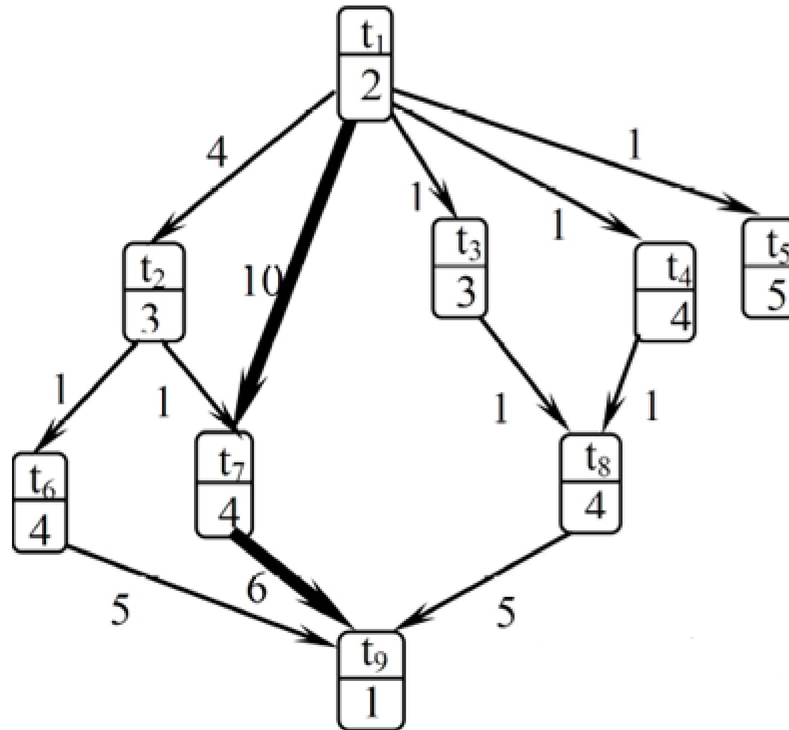


Рис. 2.1. Приклад DAG, де t_1, t_7 та t_9 є CP-вузлами

Будь-яке завдання не може бути розпочато, поки усі батьки не завершені. Нехай Pj буде процесором, на якому заплановано k-те батьківське завдання t_k завдання t_i. Час надходження даних (Data Arrival Time, DAT) t_i на процесор P_i визначається як:

$$DAT = max(FT(t_k, P_j) + c(t_i, t_k)), k = 1, 2, ..., No - parent \quad (2.1)$$

де $No - parent$ – це кількість батьків t_i,

Якщо $(i = j)$, то $c(t_i, t_k) = 0$

Батьківське завдання, яке максимізує наведений вище вираз, називається улюбленими попередниками t_i і позначається $favpred(t_i, P_j)$.

Стандартний генетичний алгоритм (Standard Genetic Algorithm, SGA) починається з початкової популяції можливого рішення. Потім, застосовуючи деякі оператори, найкращим рішенням може бути пошук через кілька поколінь. Вибір найкращого рішення визначається відповідно до значення функції

пристосованості. Відповідно до цього SGA хромосома поділяється на дві частини; відображення та планування. Розділ відображення містить індекси процесорів, на яких виконуються завдання. Розділ розкладу визначає послідовність виконання завдань. На рис. 2.2 наведено приклад такого зображення хромосоми.

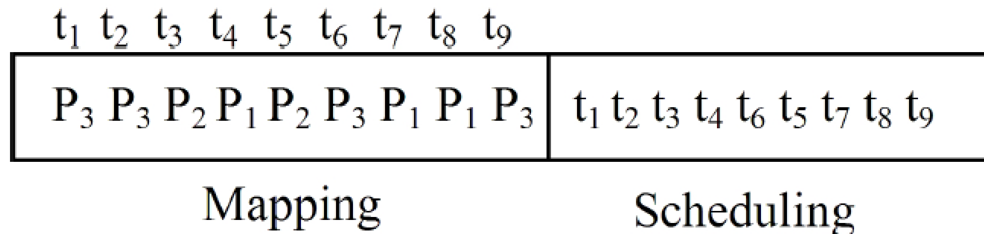


Рис. 2.2. Представлення хромосоми

Де завдання t_4, t_7, t_8 будуть заплановані на процесорі P_1, завдання t_3, t_5 будуть заплановані на процесорі P_2, а завдання t_1, t_2, t_6 і t_9 будуть заплановані на процесорі P_3. Довжина хромосоми лінійно пропорційна кількості завдань.

<u>Початкова популяція</u>

Початкова популяція будується випадковим чином. Перша частина хромосоми (тобто відображення) вибирається випадковим чином від 1 до *No-Processors*, де *No-Processors* — це кількість процесорів у системі. Друга частина (тобто розклад) генерується випадковим чином, щоб зберегти топологічний порядок графа. Псевдокод розкладу завдань із використанням SGA такий:

<u>Фітнес-функція</u>

Основна мета задачі планування полягає в тому, щоб мінімізувати довжину розкладу.

$$Fitness - Function = \left(\frac{a}{Slength}\right) \tag{2.2}$$

де *a* — константа, а *Slength* — тривалість розкладу, яка визначається наступним рівнянням:

$$Slength = max(FT[t,]), i = 1, ..., K_{noTask} \tag{2.3}$$

Function Schedule length

1. $\forall RT[P_j] = 0$ //*RT* – час готовності процесорів.

2. Нехай *LT* — список завдань відповідно до топологічного порядку DAG.

3. ***For*** *i=1* ***to*** *NoTasks* ***Do***

// *NoTasks* – це кількість завдань

(a) Видаліть перше завдання t_i зі списку *LT*.

(b) ***For*** *j = 1* ***to*** *NoProcessors* ***Do***

// *NoProcessors* — це кількість процесорів.

If t_i заплановано для процесора P_j

$ST[t_i] = max(RT[P_j], DAT(t_i, P_j))$

$FT[t_i] = ST[t_i] + weight[t_i]$

$RT[P_j] = FT[t_i]$

Endif, Endfor, Endfor.

Slength= max(FT).

Розглядаючи хромосому, представлену на рис. 2.2, як рішення DAG, представленого на рис. 2.1, функція *Fitness Time*, визначена рівнянням 2.3, була використана для обчислення тривалості розкладу (див. рис. 2.3).

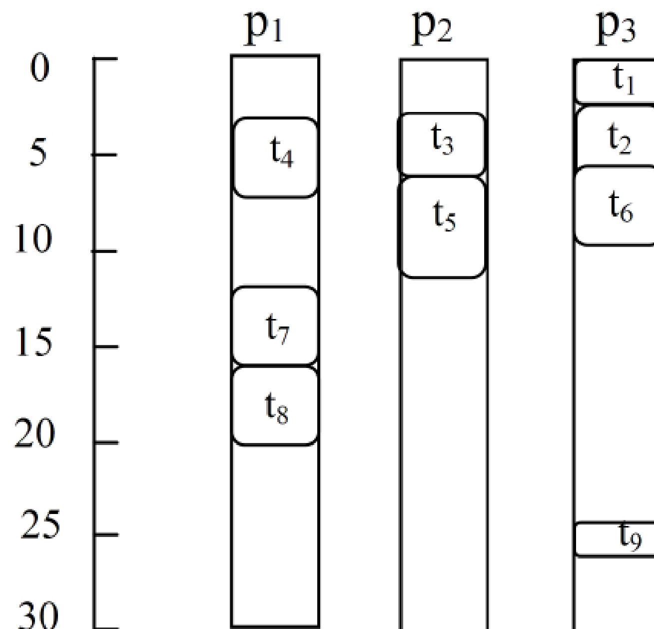


Рис. 2.3. Довжина розкладу

Генетичні оператори

Щоб застосувати оператори кросинговеру та мутації, спочатку слід застосувати фазу відбору. Ця фаза відбору використовується для розподілу репродуктивних проб між хромосомами відповідно до їх відповідності. На етапі відбору можна застосувати різні підходи. Згідно з роботою в цьому розділі, вибір колеса рулетки, пропорційний фітнес [52], і турнірний відбір [53] порівнюються таким чином, щоб використовувати найкращий метод (тобто створити найкоротший розклад). При виборі колеса рулетки ймовірність вибору пропорційна придатності хромосоми. Аналогія з колесом рулетки виникає, оскільки можна уявити, що вся популяція формує колесо рулетки з розміром слота будь-якої хромосоми, пропорційним її придатності. Потім колесо обертається, і фігурна куля вкидається. Ймовірність того, що куля зупиниться в будь-якому конкретному слоті, пропорційна дузі щілини, а отже, придатності відповідної хромосоми. У бінарному турнірному відборі дві хромосоми вибираються випадковим чином із популяції. Вибирається той, хто має вищу придатність. Цей процес повторює кількість чисельності популяції.

Відповідно до результатів дослідження [38], довжина розкладу для методу вибору турніру менша, ніж вибір колеса рулетки. Тому в кваліфікаційній роботі використовується метод турнірного відбору.

Оператор кросинговера.

Кожна хромосома в популяції піддається кросинговеру з ймовірністю μ. З популяції вибирають дві хромосоми, і для кожної хромосоми генерують випадкове число $RN \in [0, 1]$. Якщо $RN < \mu$, ці хромосоми застосовуються за допомогою одного з двох видів операторів кросинговеру: одноточковий кросинговер і оператори кросинговеру порядку. В іншому випадку ці хромосоми не змінюються. Псевдокод функції кросинговера виглядає наступним чином.

Function Crossover

1. Виберіть дві хромосоми *chrom1* і *chrom2*

2. Нехай *P* — випадкове дійсне число від 0 до 1

3. ***If*** *P <0,5* /* ймовірність операторів

Crossover-Map (chrom1, chrom2)

Else

Crossover-Order(chrom1, chrom2).

Відповідно до функції схрещування використовується один із операторів схрещування.

Карта кросинговера.

Коли вибрано єдиний кросинговер, він застосовується до першої частини хромосоми. За даними двома хромосомами генерується випадкове ціле число, яке називається точкою перетину, від 1 до *No-Tasks*. Частини хромосом, що лежать праворуч від точки кросинговеру, обмінюються для отримання двох нащадків (див. рис. 2.4).

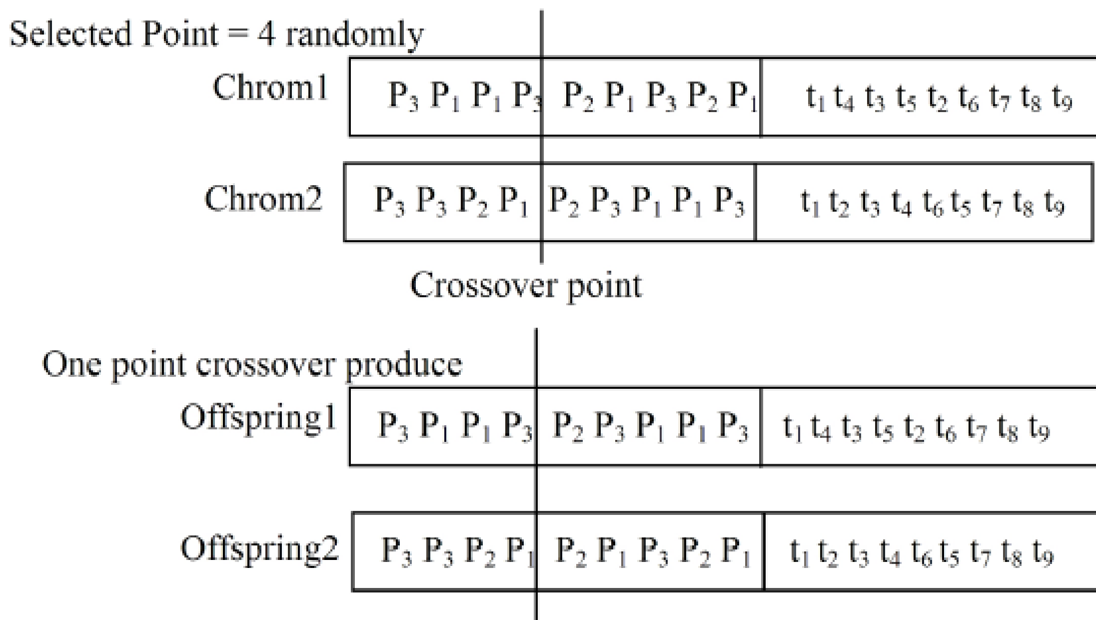


Рис. 2.4. Одноточковий кросинговер

Порядковий кросинговер.

Коли оператор порядкового кросинговеру застосовується до другої частини хромосоми, вибирається випадкова точка. Спочатку передається лівий сегмент від *chrom1* до нащадка, а потім будується правий фрагмент нащадка згідно з порядком правого сегмента оператора кросинговеру *chrom2*, наведеного на рис. 2.5.

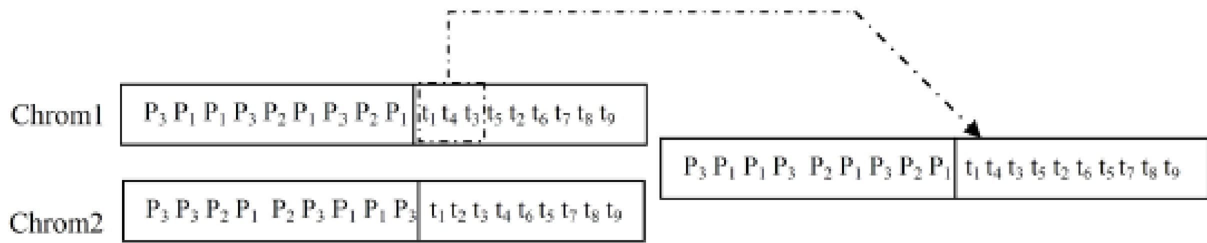


Рис. 2.5. Порядковий кросинговер

Оператор мутації.

Кожна позиція в першій частині хромосоми з ймовірністю піддається мутації. Мутація передбачає зміну призначення завдання від одного процесора до іншого. Рис. 2.6 ілюструє операцію мутації на *chrom1*.

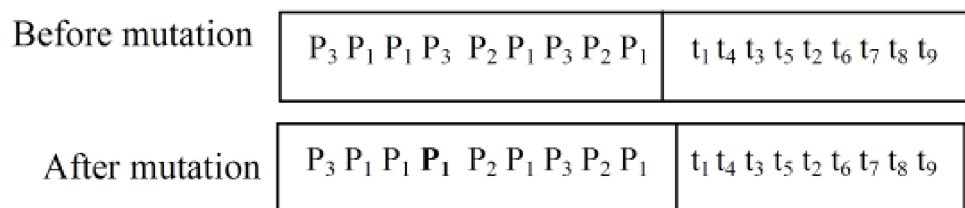


Рис. 2.6. Оператор мутації

Після застосування оператора мутації призначення t_4 змінюється з процесора P_3 на процесор P_1.

2.2 Генетичний алгоритм критичного шляху

Наступний алгоритм CPGA [38] вважається гібридом принципів GA та принципів евристичних алгоритмів (наприклад, заданий пріоритет вузлів відповідно до ALAPlevel). З іншого боку, ті самі принципи та оператори, які використовуються в алгоритмі SGA, були використані в алгоритмі CPGA. Кодування хромосоми таке ж, як і в SGA, але в початковій популяції другу частину (розклад) хромосоми можна побудувати одним із таких способів:

1. Частина розкладу будується випадковим чином, як у SGA.

2. Частина розкладу побудована за допомогою ALAP.

Function Test-Slots

1. Нехай *LT* — список готових задач

2. Спочатку список часу угоди порожній, *S-ideal-time=0* і *E-ideal-time=0*

3. Поки список *LT* не порожній, отримати завдання t_i від голови списку

(a) *Min = ST = inf*

(b) ***For*** *each processor* P_j

If t_i заплановано на P_j .

Нехай *thisST* = час початку t_i на P_j

If *thisST > MinST* ***Then*** *MinST = thisST*

If список ідеального часу P_j не порожній

For кожного часового інтервалу списку ідеального часу

If *(Eidealtime−Sidealtime) <= weight[*t_i*] & DAT(*t_i*,* P_j*) > Sidealtime*

Then запланувати t_i в ідеальний час і оновити *Sidealtime* і *Eidealtime*

Нехай *sttime* буде часом початку завдання t_i, рівним *Sidealtime*.

End If

(c) ***If*** *sttime > MinST* ***Then***

MinST = sttime.

Припустимо, графік представлений на рис. 2.3: процесор P_1 має ідеальний тайм-слот; початок цього ідеального часу (*S-ideal-time*) дорівнює 7, а його кінцевий час (*E-ideal-slot*) дорівнює 12. З іншого боку, вага *(tS)* =4 і $DAT(tS, P_1) = S-ideal-slot$ = 7. Застосувавши модифікацію, *tS* можна перепланувати початок на момент часу 7. Остаточна довжина розкладу відповідно до цієї модифікації стає 23 замість 26 (див. рис. 2.7).

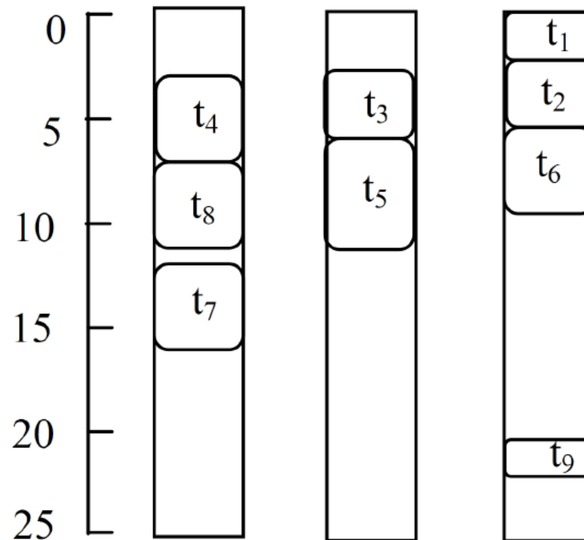


Рис. 2.7. Довжина розкладу після застосування *Function Test-Slots* скоротилася з 26 до 23

Пріоритет модифікації CPN

Відповідно до другої модифікації інший коефіцієнт оптимізації застосовується для перерахунку тривалості розкладу після надання високих пріоритетів для CPN, щоб вони могли початися якомога раніше. Цю модифікацію впроваджено за допомогою функції під назвою *Reschedule-CPNs Function*. Псевдокод цієї функції такий:

Function Reschedule-CPNs

1. Визначити CP та скласти список CPN

2. ***While*** список CPN не пустий ***DO***

- Видалити завдання t_i зі списку

- Нехай *V IP = favpred*(t_i, P_j)

If *V IP* призначено процесору P_j

Then завдання t_i призначається процесору p_j

End If

Функцію *Reschedule-CPN* була застосована у плануванні, представленому на рис.2.7. Згідно з DAG, представленим на рис. 2.1, виявлено, що CPN є t_1, t_7 і t_9. t_1 — це вузол входу, і він не має попередника, а *favpred* t_7 — це завдання t_1. Завдання t_7 заплановано для процесора P_1. Також улюбленим параметром t_9 є t_8,

але в той же час він запускається рано на процесорі P_3, тому t_9 не переміщується. Кінцева довжина розкладу зменшена до 17 замість 23 (див. рис. 2.8).

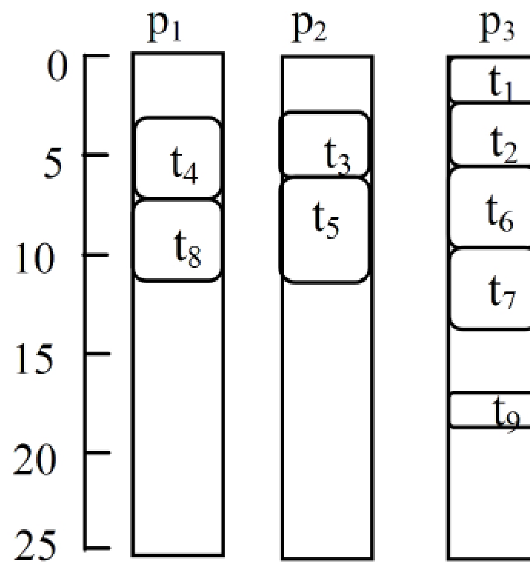


Рис. 2.8. Довжина розкладу після застосування *Function Reschedule-CPNs* зменшилась з 23 до 17

Модифікація балансу навантаження

Оскільки основною метою планування завдань є мінімізація тривалості розкладу, виявлено, що кілька рішень можуть забезпечувати однакову довжину розкладу, але баланс навантаження між процесорами може не виконуватись у деяких із них. Метою модифікації балансу навантаження є те, як отримати мінімальну довжину графіка і, в той же час, задовільнити баланс навантаження. Це було досягнуто використанням двох фітнес-функцій одна за одною замість однієї фітнес-функції. Перша фітнес-функція стосується мінімізації загального часу виконання, а друга фітнес-функція використовується для задоволення балансу навантаження між процесорами. Ця функція запропонована в [54] і розраховується як відношення максимального часу виконання (тобто тривалості розкладу) до середнього часу виконання для всіх процесорів. Якщо час виконання процесора P_j позначити через $Etime[P_j]$, то середній час виконання для всіх процесорів дорівнює:

$$avg = \sum_{j=1}^{NoProcessor} \frac{Etime[P_j]}{NoProcessors}$$

(2.4)

Отже, баланс навантаження розраховується як:

$$LoadBalance = \frac{Slength}{Avg}$$

(2.5)

На рис. 2.9 (a,b) наведено два рішення щодо планування завдань. Довжина розкладу обох рішень дорівнює 23.

Рішення a:

$$Avg = \frac{12 + 17 + 23}{3} \approx 17.33$$

$$LoadBalance = \frac{23}{17.33} \approx 1.326$$

Рішення b:

$$Avg = \frac{9 + 11 + 23}{3} \approx 14.33$$

$$LoadBalance = \frac{23}{14.33} \approx 1.604$$

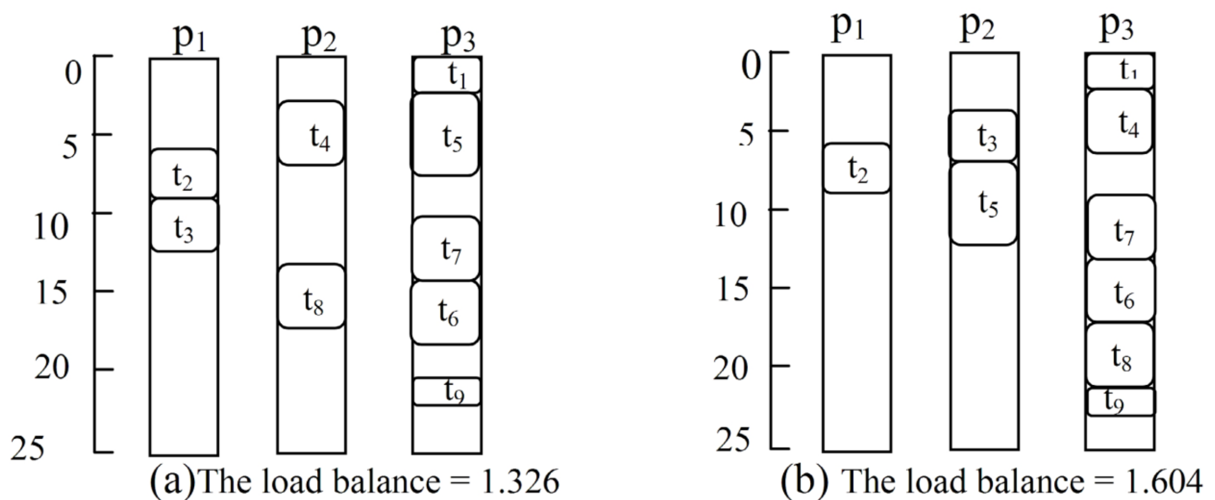


Рис. 2.9 відповідно до функції відповідності балансу рішення (а) краще, ніж рішення (б)

Згідно з функцією відповідності балансу, як показано на рис. 2.9, рішення (a) є кращим, ніж рішення (b).

Адаптивні параметри μ_c і μ_m

Автори [26] запропонували адаптивний метод для налаштування швидкості кросинговеру μ_c і частоти мутації μ_m на льоту, заснований на ідеї підтримки різноманітності в популяції без впливу на її властивості конвергенції. Тому; швидкість μ_c визначається як:

$$\mu_c = \frac{k_c(f_{max} - f_c)}{(f_{max} - f_{avg})} \tag{2.6}$$

А швидкість μ_m визначається як:

$$\mu_m = \frac{k_m(f_{max} - f_m)}{(f_{max} - f_{avg})} \tag{2.7}$$

де f_{max} є максимальним значенням відповідності, f_{avg} є середнім значенням відповідності, f_c є значенням відповідності більш підходящої хромосоми для кросинговеру, f_m є значенням відповідності хромосоми, яка підлягає мутації, k_c і k_m є позитивною реальною константою менше 1.

2.3. Генетичний алгоритм дублювання завдань

Навіть з ефективним алгоритмом планування деякі процесори можуть бути ідеальними під час виконання програми, оскільки завдання, призначені їм, можуть чекати отримання деяких даних від завдань, призначених іншим процесорам. Якби тимчасові слоти простою процесора, що очікує, можна було б ефективно використовувати шляхом визначення деяких критичних завдань і надмірного розподілу їх у цих слотах, час виконання паралельної програми можна було б ще більше скоротити [55]. Відповідно до запропонованого алгоритму [38] запропоновано хороший графік, заснований на дублюванні завдань. Цей запропонований алгоритм під назвою «Генетичний алгоритм

дублювання завдань» (TDGA) використовує генетичний алгоритм для вирішення проблеми планування.

На конкретному етапі планування для будь-якого завдання t_i на процесорі P_i, якщо $STF(favpred(t_i, p_j)) + weight(favpred(t_i, p_j)) \leq EST(t_i, p_j)$. Тоді $EST(t_i, p_j)$ можна зменшити за допомогою планування $favpred(t_i, p_j)$ до p_j. Таким чином, це визначення може бути застосоване рекурсивно вгору DAG, щоб зменшити довжину розкладу.

Вплив техніки дублювання завдань розглянуто на розкладі, представленому на рис. 2.10 (a) для DAG на рис. 2.1, довжина розкладу дорівнює 21. Якщо t_1 дублюється на процесор p_1 і p_2, довжина розкладу зменшується до 18 (див. рис. 2.10 (b)).

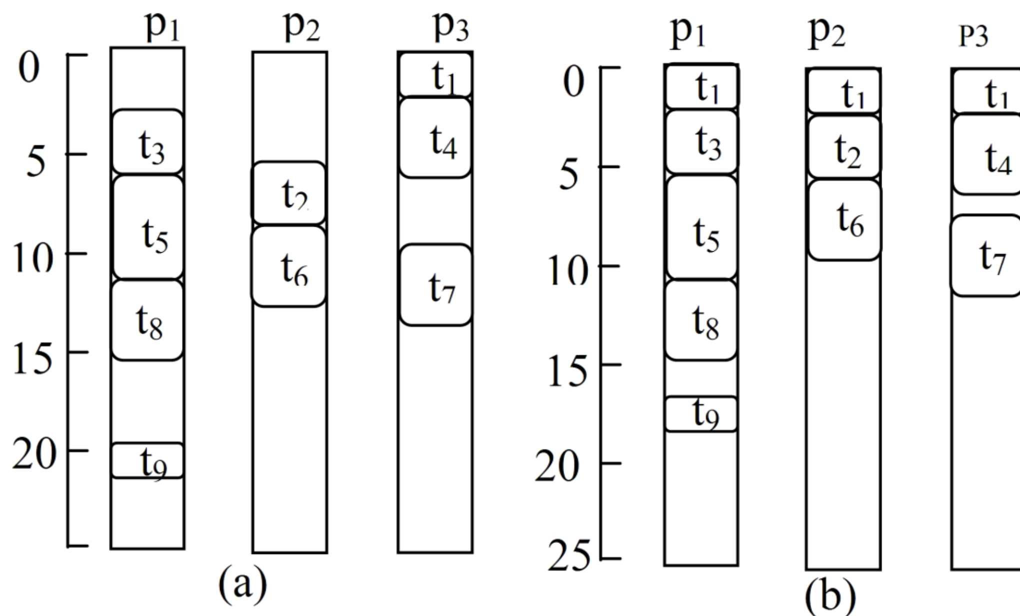


Рис. 2.10 (a) до дублювання (довжина графіка=21) (b) Після дублювання (довжина графіка=18)

<u>Загальне формулювання TDGA</u>

Згідно з алгоритмом TDGA, кожна хромосома в популяції складається з вектора пар порядку *(t, p)*, що вказує на те, що завдання *t* призначено процесору *p*. Кількість порядкових пар у хромосомі може бути різною за довжиною.

Приклад хромосоми показаний на рис. 2.11.

$$(t_2,P_1)(t_3,P_2)(t_4,P_1)(t_4,P_2)$$

Рис. 2.11. Приклад хромосоми

Перша пара порядків показує, що завдання t_2 призначено процесору P_1, а друга вказує, що завдання t_3 призначено процесору P_2 і т.д. Відповідно до принципів дублювання, одне і те ж завдання може бути призначено кілька разів різним процесорам без дублювання в одному процесорі. Якщо пара процесора завдань з’являється на хромосомі більше одного разу, розглядається лише одна з пар. Згідно з рис. 2.11, завдання t_2 призначено процесорам P_1 і P_2.

Недійсні хромосоми – це хромосоми, які не містять усіх завдань DAG. Ці недійсні хромосоми можуть бути згенеровані.

Початкова популяція.

Відповідно до алгоритму TDGA застосовується метод генерації початкової сукупності, який називається евристичним дублюванням (Heuristic Duplication, HD). Відповідно до HD, початкова популяція ініціалізується випадково згенерованими хромосомами, тоді як кожна хромосома складається точно з однієї копії кожного завдання (тобто без дублювання завдання). Потім кожне завдання випадковим чином призначається процесору. Після цього використовується техніка дублювання за допомогою функції, яка називається *Duplication-Process*. Псевдокод функції *Duplication-Process* такий:

Function Duplicatin-Process

1. Обчислити *SL* для кожного завдання в DAG

2. Скласти список завдань *Slist* відповідно до *SL* у порядку спадання

3. Взяти завдання t_i з *Slist*

4. ***While*** *Slist* не порожній.

If призначається процесору ρ_i

If $favpred(t_i, \rho_i)$ не призначено ρ_i

if $(timeslot \geq weight(favpred(t_i, \rho_i))$

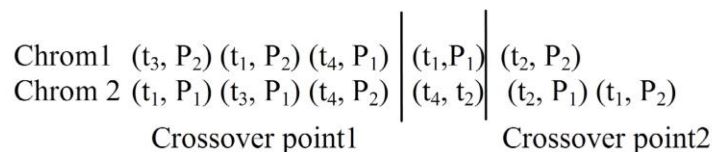
призначено $favpred(t_i, \rho_i)$ до ρ_i

Фітнес-функція.

Фітнес-функція визначається як *1/Slength*, де *Slength* визначається як максимальний час завершення всіх завдань DAG. Запропонований GA призначає нуль недійсній хромосомі як її значення придатності.

Генетичні оператори: Оператор кросинговеру.

Використовується двоточковий оператор кросинговеру. Оскільки кожна хромосома складається з векторної пари процесора завдань, кроссовер обмінюється підрядками пар між двома хромосомами. Дві точки вибираються випадковим чином, і перегородки між точками обмінюються між двома хромосомами для формування двох нащадків. Імовірність кросинговеру дає ймовірність того, що пара хромосом піддасться кросинговеру. Приклад двоточкового кросинговеру показано на рис. 2.12.



Two points 2 and 4 are generated randomly, two point crossover operator produce

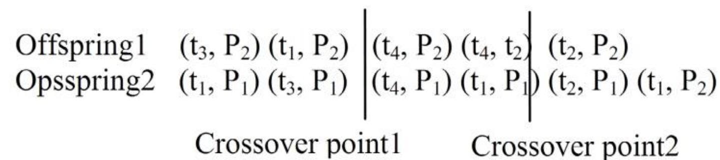


Рис. 2.12. Приклад двоточкового оператора кросинговеру.

Оператор мутації

Імовірність мутації вказує на ймовірність того, що пара порядків буде змінена. Якщо вибрано пару для мутації, номер процесора цієї пари буде змінено випадковим чином. Приклад оператора мутації показано на рис. 2.13.

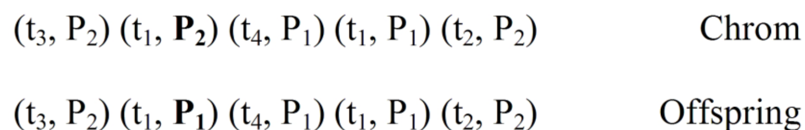


Рис. 2.13. Приклад оператору мутації

2.4. Генетичний алгоритм з обмеженнями

У більшості застосувань GA для задач оптимізації з обмеженнями використовується метод штрафної функції. У методі штрафної функції для обробки обмежень нерівності в задачах мінімізації функція відповідності $F(x\to)$ визначається як сума цільової функції $f(x\to)$ і штрафного значення, який залежить від порушення обмеження $\langle g_j(x\to)\rangle$

$$F(x\to) = f(x\to) + \sum_{j=1}^{J} R_j \langle g_j(x\to)\rangle^2, \tag{2.8}$$

де $\langle\rangle$ позначає абсолютне значення операнда, якщо операнд від'ємний і повертає нульове значення в протилежному випадку. Параметр R_j є параметром штрафу j-го обмеження нерівності. Мета штрафного параметра R_j полягає в тому, щоб зробити порушення обмеження $g_j(x\to)$ такого ж порядку, що й значення цільової функції $f(x\to)$. Обмеження рівності зазвичай обробляються шляхом перетворення їх на обмеження нерівності наступним чином:

$$g_{k+J}(x\to) \equiv \delta - |h_k(x\to)| \geqslant 0, \tag{2.9}$$

де δ – невелике позитивне значення. Це збільшує загальну кількість обмежень нерівності до $m=J+K$ і до члена J у рівнянні. Потім (2.8) можна замінити на m, щоб включити всі нерівності та обмеження рівності. Таким чином, існує всього m штрафних параметрів R_j, які повинні бути встановлені правильно в підході штрафної функції.

Щоб зменшити кількість штрафних параметрів, часто обмеження нормалізують і використовують лише один штрафний параметр R [57]. У будь-якому випадку, є дві проблеми, пов'язані з цим підходом до статичної функції штрафу:

1. Оптимальний розв'язок $F(x\to)$ залежить від штрафних параметрів R_j (або R). Користувачам зазвичай доводиться пробувати різні значення R_j (або R),

щоб знайти значення, яке спрямує пошук до можливого регіону. Це вимагає масштабних експериментів, щоб знайти будь-яке розумне рішення. Ця проблема настільки серйозна, що деякі дослідники використовували різні значення R_j (або R) залежно від рівня порушення обмежень [58], а деякі використовували складну температурну еволюцію штрафних параметрів через покоління [59], що включає кілька параметрів, які описують швидкість еволюції.

2. Включення параметру штрафу спотворює цільову функцію [57]. Для малих значень R_j (або R) викривлення є невеликим, але оптимум $F(x\rightarrow)$ може бути не близьким до справжнього обмеженого оптимуму. З іншого боку, якщо використовується великий R_j (або R), оптимум $F(x\rightarrow)$ буде ближчим до справжнього обмеженого оптимуму, але викривлення може бути настільки серйозним, що $F(x\rightarrow)$ може мати штучний локальний оптимальний рішення. Це в першу чергу відбувається через взаємодію між кількома обмеженнями. Щоб уникнути таких локально оптимальних рішень, класичний підхід до функції штрафу працює в послідовностях, де в кожній послідовності параметри штрафу збільшуються поетапно, а поточна послідовність оптимізації починається з оптимізованого рішення, знайденого в попередній послідовності. Таким чином можливий контрольований пошук і можна уникнути локально оптимальних рішень. Однак більшість класичних методів використовують методи пошуку на основі градієнтів і зазвичай мають труднощі при розв’язанні проблем із дискретним простором пошуку та проблем із великою кількістю змінних. Хоча GA не використовують градієнтну інформацію, вони не позбавлені ефекту спотворення, викликаного додаванням штрафного терміну з цільовою функцією. Однак GA порівняно менш чутливі до спотворених ландшафтів функцій через стохастичність їхніх операторів.

При застосуванні GA з підходом штрафної функції до задач обмеженої оптимізації користувачеві зазвичай доводиться виконувати багато прогонів або «відкоригувати» параметри штрафу, щоб отримати рішення в межах прийнятної межі.

Міхалевич [60], а пізніше Міхалевич і Шонауер [61] обговорили різні методи обробки обмежень, що використовуються в GA. Вони класифікували більшість еволюційних методів обробки обмежень на п'ять категорій:

1. Методи, засновані на збереженні здійсненності рішень;
2. Методи, засновані на штрафних функціях;
3. Методи розрізнення здійсненних і нездійсненних рішень;
4. Методи на основі дешифраторів;
5. Гібридні методи.

Методи першої категорії явно використовують знання структури обмежень і використовують оператор пошуку, який підтримує здійсненність рішень. Другий клас методів використовує штрафні функції різних типів, включаючи динамічні штрафні підходи, де штрафні параметри адаптуються динамічно з часом. Третій клас методів обробки обмежень використовує різні оператори пошуку для обробки нездійсненних і здійсненних рішень. Четвертий клас методів використовує непряму схему представлення, яка містить інструкції для побудови можливого рішення. У п’ятій категорії еволюційні методи поєднуються з евристичними правилами або класичними методами обмеженого пошуку. Міхалевич і Шонауер [61] порівняли різні алгоритми для ряду тестових завдань і помітили, що кожен метод добре працює на деяких класах задач, тоді як погано працює на інших задачах. Через цю неузгодженість у виконанні різних методів вони запропонували використовувати метод статичної функції штрафу, подібний до того, що наведено в рівнянні (2.8). Останнім часом розроблено метод двофазного еволюційного програмування (ЕП) [63]. На першому етапі була використовується стандартна техніка ЕП з низкою параметрів стратегії, які були розроблені в процесі оптимізації. З рішенням, отриманим на першому етапі, на другому етапі використовується метод нейронної мережі для покращення рішення. Продуктивність другої фази залежить від того, наскільки близьке рішення до справжнього оптимального рішення знайдено на першій фазі. Цей підхід включає занадто багато різних процедур із багатьма контрольними параметрами, і незрозуміло, які процедури та налаштування параметрів важливі.

Незрозуміло, як цей досить складний метод збільшить свою ефективність для більш складних проблем.

Запропонований у [56] метод належить як до другої, так і до третьої категорій методів обробки обмежень, описаних Міхалевичем і Шонауером [61]. Хоча штрафний термін додається до цільової функції, щоб покарати нездійсненні рішення, метод відрізняється від способу визначення штрафного терміну в звичайних методах і в попередніх реалізаціях GA.

Метод пропонує використовувати оператор вибору турніру, де одночасно порівнюються два рішення, і завжди застосовуються такі критерії [64]:

1. Будь-яке можливе рішення є кращим перед будь-яким нездійсненним рішенням.

2. Серед двох можливих рішень надається перевага тому, що має краще значення цільової функції.

3. Серед двох нездійсненних рішень надається перевага тому, що має менше порушення обмежень.

Хоча існує низка інших реалізацій [60], [62], [65], де критерії, подібні до наведених вище, накладаються на їхні підходи до обробки обмежень, усі ці реалізації використовували різні показники порушень обмежень, які все ще потребували штрафного параметра для кожне обмеження.

Штрафні параметри необхідні, щоб зробити значення порушення обмежень того ж порядку, що й значення цільової функції. У запропонованому методі штрафні параметри не потрібні, оскільки в жодному з трьох вищевказаних сценаріїв рішення ніколи не порівнюються з точки зору цільової функції та інформації про порушення обмежень. Крім того, ідея порівняння нездійсненних рішень лише з точки зору порушення обмежень має практичний підтекст. Щоб оцінити будь-яке рішення, звичайною практикою є спочатку перевірка здійсненності рішення. Якщо рішення є нездійсненним (тобто принаймні одне обмеження порушено), розробник ніколи не потрудиться обчислити значення цільової функції (наприклад, вартість проекту). Немає сенсу обчислювати

значення цільової функції нездійсненного рішення, оскільки рішення просто не може бути реалізоване на практиці.

В наступній функції відповідності нездійсненні рішення порівнюються лише на основі їх порушення обмежень:

$$F(x \to) = f(x \to) \ if \ \ g_j(x \to) \geq 0 \ \ \forall j = 1,2,\dots,m, \ f_{max} + \sum_{j=1}^{m} \langle g_j(x \to) \rangle \, otherwise \qquad (2.10)$$

Параметр f_{max} є значенням цільової функції найгіршого можливого рішення в сукупності. Таким чином, придатність нездійсненного рішення залежить не тільки від кількості порушень обмежень, а й від сукупності доступних рішень. Однак придатність можливого рішення завжди фіксована і дорівнює значенню його цільової функції.

Спочатку ця техніку обробки обмежень буде проілюстрована на задачі обмеженої мінімізації з однією змінною, а пізніше показано її вплив на контури двовимірної проблеми. На рис. 2.13 показано функцію пристосування *F(x→)* (товста суцільна лінія в недопустимій області та пунктирна лінія у допустимій області). Необмежене мінімальне рішення тут неможливе. Важливо відзначити, що *F(x→)=f(x→)* у допустимій області, і існує поступове збільшення придатності для нездійсненних рішень далеко від межі обмежень. Відповідно до згаданого раніше оператора відбору турнірів, буде здійснюватися вибірковий тиск на те, щоб нездійсненні рішення були наближені до можливого регіону. На малюнку також показано, як буде оцінюватися значення фізичної форми шести членів популяції (показано суцільними маркерами). Цікаво відзначити, як придатність нездійсненних рішень залежить від найгіршого можливого рішення. Якщо в сукупності не існує можливого рішення, f_{max} встановлюється рівним нулю.

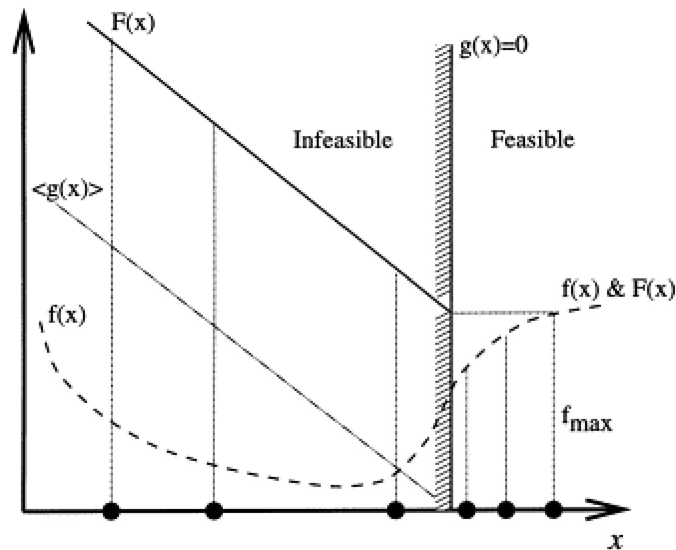


Рис. 2.13. Схема обробки обмежень (на осі x відмічені рішення у популяції GA)

Оскільки рішення не порівнюються як за значенням цільової функції, так і за інформацією про порушення обмежень, у запропонованому методі немає потреби в будь-якому явному параметрі штрафу. Це головна перевага даного методу в порівнянні з попередніми реалізаціями штрафної функції з використанням GA. Однак, щоб уникнути будь-яких зміщень від будь-якого конкретного обмеження, усі обмеження нормалізуються (звичайна практика оптимізації з обмеженнями [57]) і використовується рівняння. (2.10). Важливо відзначити, що така схема обробки обмежень без необхідності штрафного параметра можлива, оскільки GA використовують популяцію рішень у кожній ітерації, а попарне порівняння рішень можливе за допомогою оператора вибору турніру. З цієї ж причини такі схеми не можна використовувати з класичними методами поточкового пошуку та оптимізації.

Даний метод дещо схожий на метод PS [62], який передбачає штрафні параметри. Таким чином, як і інші підходи до функції штрафу, метод PS також чутливий до параметрів штрафу. Крім того, метод PS іноді може створювати штучні локальні оптимуми. Процедура розрахунку функції однієї змінної, показаної на рис. 2.13, через фітнес-функцію в методі PS проілюстрована на рис. 2.14.

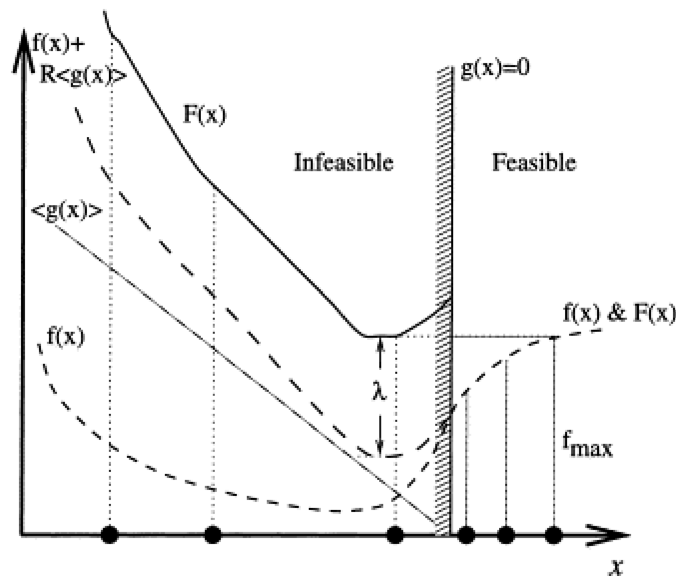


Рис. 2.14. Проілюстровано схему обробки обмежень Пауелла та Сколніка (на осі x відмічені рішення у популяції GA)

Основна відмінність методу PS від запропонованого полягає в тому, що в методі PS значення цільової функції враховується при розрахунку придатності нездійсненних рішень. У методі PS штрафне значення функції $f(\vec{x})+R\sum_j \langle g_j(\vec{x}) \rangle$ підвищується на величину λ (показано на рисунку), щоб зробити придатність найкращого нездійсненного рішення рівною придатності найгіршого можливого рішення. На рис. 2.14 показано, що в певних ситуаціях результуюча функція відповідності (показана довгою пунктирною лінією) може мати штучний мінімум у нездійсненній області. Коли можлива область вузька, у сукупності може бути небагато можливих рішень. У такому випадку GA з цим методом обробки обмежень можуть потрапити в пастку цього штучного локального оптимуму. Варто зазначити, що ефект цього штучного локального оптимуму може бути зменшений, якщо використовувати досить великий штрафний параметр *R*. Ця залежність методу обробки обмежень від параметра штрафу є небажаною (а значення «великого параметра штрафу» є суб’єктивним для проблеми, що розглядається) і часто змушує дослідників повторно запускати алгоритм оптимізації з різними значеннями параметрів штрафу.

У оптимізації з обмеженнями реальних параметрів із використанням GA схеми, що визначають суміжні області в просторі пошуку, можуть вважатися більш важливими, ніж схеми, що визначають дискретні області в просторі пошуку. У двійковій GA під одноточковим оператором кросинговеру всі загальні схеми, що відповідають обом батьківським рядкам, зберігаються в обох дочірніх рядках. Оскільки будь-яка довільна безперервна область у просторі пошуку не може бути представлена за допомогою єдина схема Холланда, і оскільки можливий простір пошуку зазвичай може мати будь-яку довільну форму, очікується, що оператор одноточкового перетину, який використовується в бінарних GA, не завжди зможе створити можливі дочірні рішення з двох можливих батьківських рішень. У більшості випадків такі проблеми мають можливу область, яка становить крихітну частку всього простору пошуку. Таким чином, як тільки можливі вихідні рішення знайдені, потрібний керований оператор перехрещення, щоб створювати дочірні рішення, які також є здійсненними.

Представлення змінних із плаваючою комою в GA та оператор пошуку, який поважає безперервні області в просторі пошуку, можуть усунути дві вищезазначені труднощі, пов’язані з двійковим кодуванням і одноточковим кросовером.

Існує кілька способів збереження різноманітності в популяції. Серед них популярні методи ніші [66] і використання мутації [67]. Переважно використовують один або обидва вищезазначені методи збереження різноманітності серед можливих рішень. В операторі вибору турніру реалізована проста стратегія ніші. При порівнянні двох можливих рішень (i та j) між ними вимірюється нормалізована евклідова відстань d_{ij}. Якщо ця відстань менша за критичну відстань $\bar{d}$, розв’язки порівнюються зі значеннями цільової функції. В іншому випадку вони не порівнюються і перевіряється інше рішення j. Якщо перевіряється конкретна кількість (n_f) можливих рішень і не знайдено жодного, що відповідає вимогам критичної відстані, i-те рішення оголошується переможцем. Нормована евклідова відстань обчислюється наступним чином:

$$d_{ij} = ln \sum_{k=1}^{n} x_k^{(i)} - x_k^{(j)} x_k^{u} - x_k^{l2}$$

(2.11)

Таким чином, рішення, які знаходяться далеко одне від одного, не порівнюються, і можна зберегти різноманітність серед можливих рішень.

2.5 Висновки

У другому розділі були розглянуті приклади використання інструментів, обраних у першому розділі, та прийняте остаточне рішення щодо особливостей реалізації алгоритмів розв’язання задачі.

Була представлена реалізація стандартного GA для вирішення проблеми планування завдань. Особливістю є використання хромосомного представлення для розкладу занять. До цього SGA було додано деякі модифікації, щоб покращити продуктивність планування. Ці модифікації базуються на об’єднанні евристичних принципів із принципами GA. Алгоритм під назвою «Генетичний алгоритм критичного шляху» заснований на переплануванні вузлів критичного шляху у хромосомі, а потім у різних поколіннях. Другий розроблений генетичний алгоритм, «Генетичний алгоритм дублювання завдань», заснований на техніці дублювання завдань для подолання накладних витрат на комунікації.

Також у запропонованій методиці рішення ніколи не порівнюються як за значенням цільової функції, так і за інформацією про порушення обмежень. Нездійсненні рішення штрафуються таким чином, щоб забезпечити напрямок пошуку до можливого регіону, і коли знаходять адекватні можливі рішення, використовується схема нішування для підтримки різноманітності, що допомагає оператору кросинговеру GA знаходити все кращі рішення з генерацією. Це стало можливим головним чином завдяки популяційному підходу GA та можливості попарного порівняння рішень за допомогою оператора вибору турніру.

РОЗДІЛ 3

РОЗРОБКА МЕТОДИКИ ЗАСТОСУВАННЯ ГЕНЕТИЧНОГО АЛГОРИТМУ ДО РОЗВ'ЯЗАННЯ ЗАДАЧІ СКЛАДЕННЯ РОЗКЛАДУ ЗАНЯТЬ УНІВЕРСИТЕТУ

3.1. Обґрунтування вибору алгоритму

Як було зазначено у попередньому розділі, для складання розкладу був обраний генетичний алгоритм. Фактичною альтернативою йому можуть бути або інші перебірні алгоритми, скажімо, динамічне програмування, або алгоритми машинного навчання, такі, як нейронні мережі типу LSTM. Проте в цій роботі був обраний саме генетичний алгоритм, оскільки методи динамічного програмування через складність вхідних даних дадуть порівняно більшу складність коду, а щодо нейромереж є обмеження у вигляді відсутності значного обсягу підготованих даних для навчання та значний час навчання у випадку зміни характеру обмежень, що накладаються на задачу.

Будь-який алгоритм за визначенням має три типи складності: часову, просторову та складність коду. Генетичний алгоритм належить до класу NP-повних задач, для яких часова складність є експоненційною величиною і може бути необмежено великою. Тож, перед тим, як почати обчислення, варто виконати оцінку, чи можливо взагалі знайти рішення за розумний час.

Відповідь на питання щодо обчислюваності алгоритму для комбінаторних алгоритмів є відкритою. Тим паче, що на знайдене рішення можуть бути накладені певні вимоги та обмеження щодо якості цього рішення. Зокрема, стосовно генетичного алгоритму можуть бути висунуті такі вимоги та обмеження як до окремих особин, так і до популяції загалом. Тому на обчислюваність генетичного алгоритму впливають як характер геному (інформація, що міститься у генах), так і характер обмежень, що накладаються на знайдене рішення.

З огляду на сказане вище, задача складання розкладу для навчального закладу виглядає так: навчальний заклад використовує ресурси викладацького складу, аудиторного фонду та часу для навчання студентських груп. Маючи відомості про стан цих ресурсів, а також про порядок викладання курсів та ліміти часу, виділені на таке викладання, треба скласти розклад занять. При складанні можуть бути враховані критерії оптимізації, наприклад, максимально щільне завантаження аудиторій, найменша дистанція, яку мають долати студенти під час зміни аудиторій між парами тощо. Крім оптимізаційних критеріїв також треба врахувати обмеження різного роду. Ці обмеження необхідні, щоб врахувати наявність вільних аудиторій необхідної місткості та обладнаних відповідним чином, можливість об'єднання студентських груп у потоки, наявність у викладача попередньо набутого досвіду читання запропонованого курсу, наявність одного лектора на лекціях та двох викладачів на практичних заняттях тощо.

Метою роботи генетичного алгоритму побудови розкладу є складання такого розкладу, де всі обов'язкові для вивчення навчальні курси мають бути обов'язково викладені всім групам, які повинні їх прослухати. При цьому бажано врахувати якомога більше вимог та обмежень, зважаючи на рівень їхнього пріоритету. Треба мати на увазі, що через велику кількість параметрів для складання комбінацій, та високу варіативність деяких з них, обчислювальна складність такого алгоритму може бути дуже високою. Це означатиме дуже, а часом, навіть, аж занадто тривалі розрахунки. Тому варто врахувати ліміт часу або кількості поколінь для генерації розкладу. Генетичний алгоритм, що складатиме розклад, має як мінімум одне прийнятне рішення за умови, що обмеження не погіршують генетичну інформацію таким чином, що критично необхідні її елементи не стають взаємовиключними.

3.2. Вхідні дані

Як зазначалося, вхідними даними для задачі складання розкладу є навчальний план, перелік студентських груп, запланованих для навчання, відомості про штатний розклад, дані про аудиторний фонд та перелік обмежень.

3.2.1 Навчальний план.

Навчальний план є джерелом генетичної інформації та фактично основою ключового обмеження, воно полягає в тому, що в результаті еволюції генетичний алгоритм має знайти особину - розклад, яка задовольняє тій умові, що всі курси, які мають бути викладені кожній студентській групі мають бути відображені у розкладі. Навчальний план складається з опису курсів для кожної спеціальності, де в інформації зазначається назва курсу, семестр або семестри викладання, чверть або чверті викладання, кількість лекційних занять, необхідне для проведення лекцій обладнання, кількість лабораторних занять, необхідне обладнання для проведення лабораторних занять, наявність курсового проекту та кількість годин, відведена на його виконання. Приклад об'єкта вхідних даних для навчального плану наведений у лістингу 3.1.

Лістинг 3.1. Об'єкт навчальної дисципліни

```
{
    "name": "Основи програмування",
    "semesters": [1, 2],
    "quarters": [1, 2, 3, 4],
    "lectures": { "hours": 74, "equipment": [] },
    "labs": { "hours": 100, "equipment": ["комп'ютер"] },
        "project": {}
 }
```

3.2.2 Перелік груп.

Цей набір даних також є частиною найважливішого обмеження. Він складається за напрямами та містить інформацію про кількість студентів у кожній групі станом на початок навчального року. Приклад об'єкта для переліку груп наведений у лістингу 3.2.

Лістинг 3.2. Об'єкт студентської групи:

```
{
    'year': 2022,
    'name': '121 22-1',
    'quantity': 20,
    'speciality': '121',
    'type': 'група'
 }
```

3.2.3 Навчальне навантаження

Для того, щоб визначити викладача певного курсу для певної групи, треба знати штатний розклад викладачів по окремих кафедрах. При чому, під час складання розподілу навантаження, треба враховувати поточне навантаження, щоб знайдене генетичним алгоритмом рішення або збігалося з розрахованим навантаженням, або було максимально близьким до нього.

Лістинг 3.3. Приклад об'єкта для навчального навантаження:

```
{
        "surname": "Прізвище",
        "first_name": "Ім'я",
        "second_name": "По батькові",
        "degree": "доктор філософії",
        "position": "доцент",
        "courses": [
                        {
                            "name": "",
```

```
                        "lectures":
                        "labs": ...,
                        "project": ...
                    }
                ]
        }
```

3.2.4 Аудиторний фонд

Для складання розкладу занять необхідно визначитись з тим, якими навчальними аудиторіями можна скористатись для проведення окремих видів занять з огляду на місткість аудиторії та тип наявного у ній обладнання для проведення тих чи інших спеціалізованих видів навчання.

Приклади об’єктів для аудиторного фонду наведено у лістингу 3.4.

Лістинг 3.4. Приклад об’єкта аудиторного фонду:

```
{
        "number": "1",
        "type": "лекційна",
        "subtype": "загального призначення",
        "size": "60",
        "equipment": [],
    },
    {
        "number": "11",
        "type": "лабораторія",
        "subtype": "комп'ютерна",
        "size": "20",
        "equipment": ["комп'ютери"],
    }
```

3.2.5 Обмеження

Цей набір даних є надважливим, оскільки він фактично визначає якість популяції на кожному кроці, та якість особини - переможниці в решті решт. Для обмежень, які не припускають варіанту не виконання (обов'язкові обмеження) треба встановити найвищий пріоритет. Набір обмежень допускає існування купи обмежень з однаковим пріоритетом.

Приклад об'єктів обмежень наведений у лістингу 3.5:

```
{
        "priority": 100,
        "course": "all",
        "group": "all"
    },
    {
        "priority": 99,
        "course": "all",
        "equipment": "all"
    },
    {
        "priority": 98,
        "course": "all",
        "tutor": "all"
    }
```

3.3. Опис алгоритму

Перед початком написання програми, що реалізує занадто складні за часом алгоритми, треба визначитись, чи варто взагалі починати обчислення. Це питання має не метафізичний, але цілком практичний вимір, оскільки можна спробувати математично визначитись, чи є можливість отримати результат за розумний проміжок часу чи ні. Це називається обчислюваністю алгоритму. Щодо поставленої задачі, спочатку треба перевірити, чи не є висунуті обмеження

несумісними. Якщо це не так, то варто шукати рішення в будь-якому разі, при чому бюджет часу на пошук варто встановити таким, щоб він збігався у часі зі складанням такого розкладу вручну. Звісно, якщо рішення існує, обчислювальна машина знайде його значно швидше, ніж людина.Але через спосіб складання, людина одразу отримає якісніший розклад і потенційно може виявитись, що за означений час машина знайде цілу купу можливих варіантів розкладу, які за якістю (кількість врахованих обмежень) будуть гірші ніж рішення, обчислювані людиною.

Популяція та хромосоми

Для успішної роботи генетичного алгоритму необхідно правильно скласти уявлення про вміст хромосом окремих особин та розмір популяції. Те, наскільки зручно або незручно буде представлена інформація у хромосомі, може суттєво вплинути на швидкість роботи програми. Кількість особин у популяції впливає на час, що потребує алгоритм для знаходження правильного або найкращого рішення. Також необхідно визначитись, чи є кількість хромосом в особини змінною, чи ні. Це впливає на складність коду.

Для розв'язання поставленої задачі у цій роботі був запропонований підхід, де хромосомою є інформація про певну пару, особиною є потижневий розклад на цілий рік, кількість хромосом в особині незмінна, оскільки кожного робочого дня може відбуватися пари з 1 по 8. Цей алгоритм жодним чином не враховує перенесення занять з робочих днів тижня на вихідні. Завдяки випадковому формуванню генів та за умови досить великої популяції досягається розмай генетичної інформації що позитивно впливає на збіжність алгоритму та якість кінцевого рішення. Кількість особин в популяції протягом роботи алгоритму не зменшується.

Запропонований такий вигляд хромосоми у вигляді хеш-таблиці:

```
{
        "weektype": "...",
        "dow": "...",
        "clsNum": x,
```

```
        "audience": "...",
        "tutors": [...],
        "quarters": [],
        "type": "...",
        "equipment": [],
        groups: [],
    }
```

У наведеній хромосомі гени мають таке тлумачення:

`dow (day of week)` - день тижня (понеділок, вівторок, середа, четвер, п'ятниця);

`clsNum` - номер пари (від 1 до 8);

`audience` - номер аудиторії / лабораторії;

`tutor` - викладач;

`quarter` - навчальна чверть (від 1 до 4);

`type` - тип заняття (лекція / практика);

`equipment` - обладнання аудиторії

`groups` - студентські групи на цій парі

Генерація початкової популяції.

Генерація початкової популяції відбувається шляхом випадкового заповнення усіх хромосом усіх особин з урахуванням вимог навчального плану (номер чверті чи семестру, кількість годин на тиждень тощо). Застосування обов'язкових, а, тим більше необов'язкових обмежень на цьому етапі не суттєве, оскільки в цій роботі використана повноцінна еволюція з усіма її елементами: природним відбором, схрещуванням та мутацією.

Природний відбір.

З двох найпопулярніших методик природного відбору: турнірного відбору та випадкового відбору, в кваліфікаційній роботі була використана методика турнірного відбору. Це обґрунтовано тим, що спочатку треба знайти найменш пошкоджені особини, не втрачаючи різноманіття генів. Турнірний відбір забезпечує такий перебіг подій, а випадковий відбір ні. Потім на кожному еволюційному кроці виконувались відбори особин для формування наступного

покоління, шляхом накладання всіх видів обмежень. При цьому повне початкове здоров'я кожної особини було прийнято за 100 балів, кожна невідповідність обов'язковим обмеженням каралася штрафом у 10 балів, а кожна невідповідність необов'язковим обмеженням - штрафом у 5 балів.

Схрещування.

Цей еволюційний механізм у цій роботі був застосований у двох варіантах: половинним склеюванням та гребінкою. Половинне склеювання означає формування нової особини з першої половини генів першого батька та другої половини генів другого батька. Схрещування гребінкою означає формування нової особини з парних за ліком генів одного батька та непарних генів іншого. Таким чином, кожна операція схрещування формує з двох батьків дві нові особини. Таким чином, після схрещування кількість особин у популяції залишається сталою.

Мутація.

Для того, щоб у складеному розкладі були враховані всі студентські групи та курси, і якомога більша кількість штатних викладачів (а також інші вимоги), треба забезпечувати розмай значень генів. Але природний відбір, випадковість під час початку еволюції не забезпечують наявність усіх можливих значень генів між особинами популяції. Тому в цій роботі був використаний механізм мутацій, який дозволяє урізноманітнити генофонд поколінь. Були досліджені три варіанти кількості мутованих генів з метою визначення найкращого результату.

3.4. Дослідження алгоритму

В ході дослідження складеного генетичного алгоритму, були випробувані різні значення що могли вплинути на швидкість та збіжність генетичного алгоритму.

По-перше, були використані популяції різного розміру (50, 100, 150 та 200 особин). Такі розміри популяцій для генетичного алгоритму не є аж надто

великими, але через велику кількість хромосом кожна особина займає досить великий обсяг пам’яті, і розрахунки для завеликої популяції вимагають неабияких обчислювальних ресурсів. Тому були обрані середні за розміром значення.

Для початку була досліджена поведінка генетичного алгоритму у популяціях з наведеними кількостями особин, за умови класичного турнірного відбору з двох особин, схрещуванням половинним склеюванням та кількістю мутованих генів у 5%. Результати експерименту наведені на рис. 3.1.

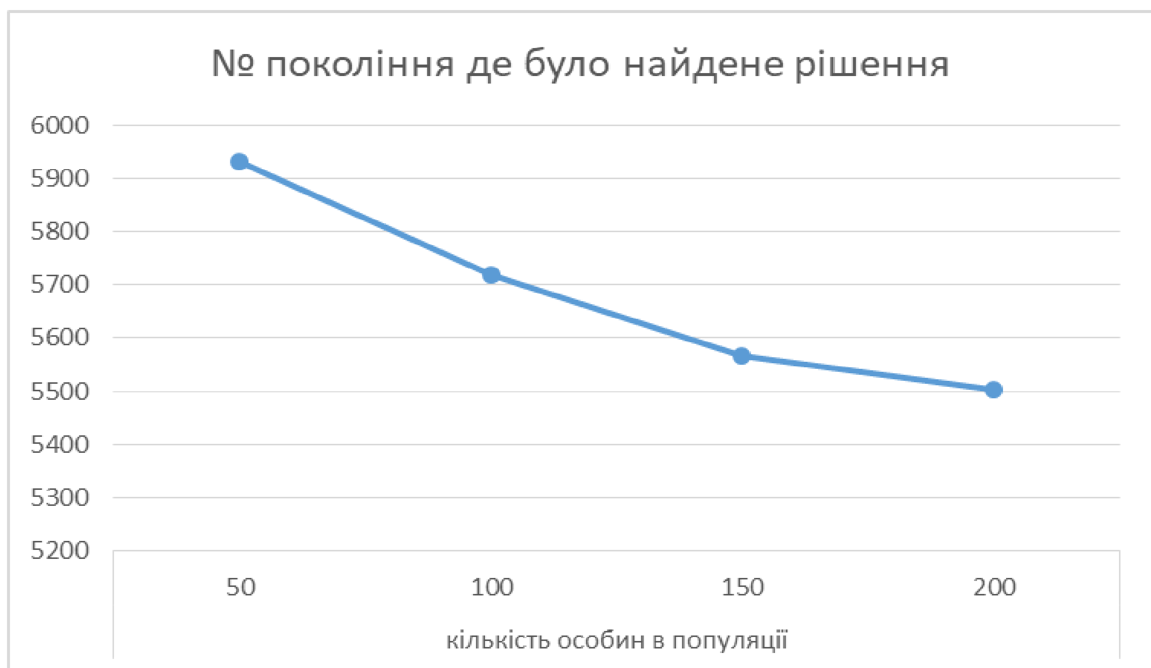


Рис. 3.1. Результати пошуку рішення генетичного алгоритму з турніром на дві особини схрещуванням половинним склеюванням та мутацією 5%.

Надалі проводилися випробування, аби з’ясувати ступінь впливу кожного з параметрів та найкраще значення кожного з них.

Турнірний відбір часто роблять на кілька особин. Зазвичай до певної міри це скорочує час обчислень, але після певного збільшення кількості учасників турніру, часові результати роботи алгоритму суттєво погіршуються. Для знаходження оптимальної кількості учасників турніру були виконані експерименти с трьома, чотирма та п'ятьма учасниками. Відповідні результати наведені на рис. 3.2, 3.3 та 3.4.

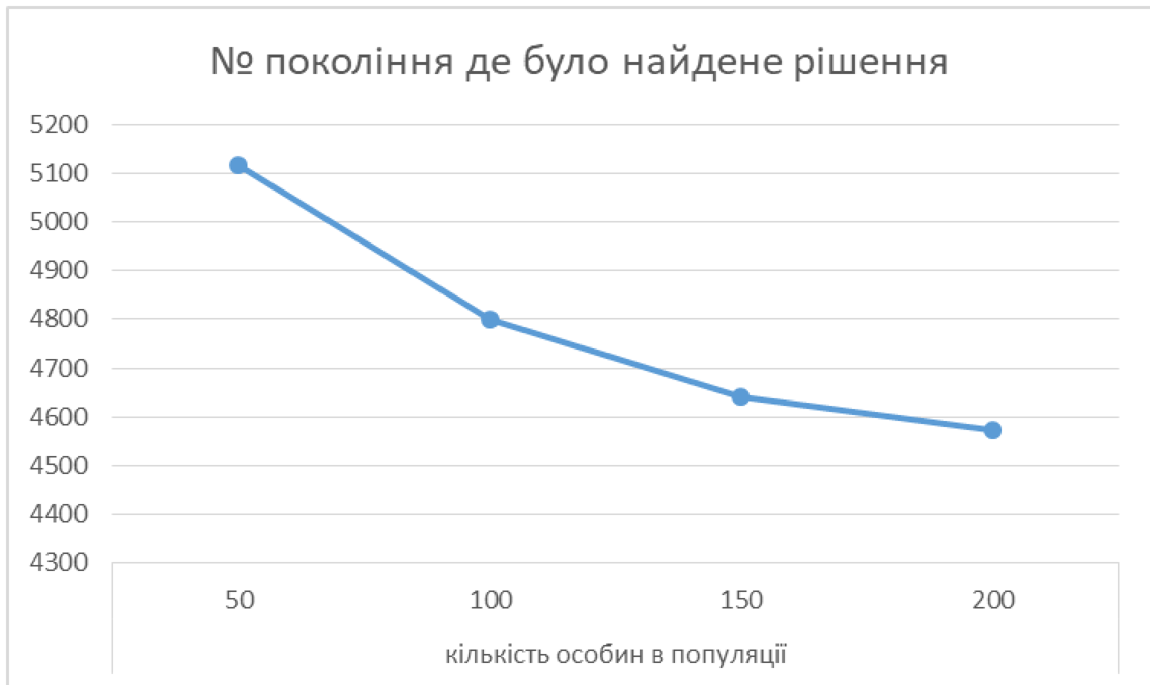


Рис. 3.2. Результати пошуку рішення генетичного алгоритму з турніром на три особини схрещуванням половинним склеюванням та мутацією 5%.

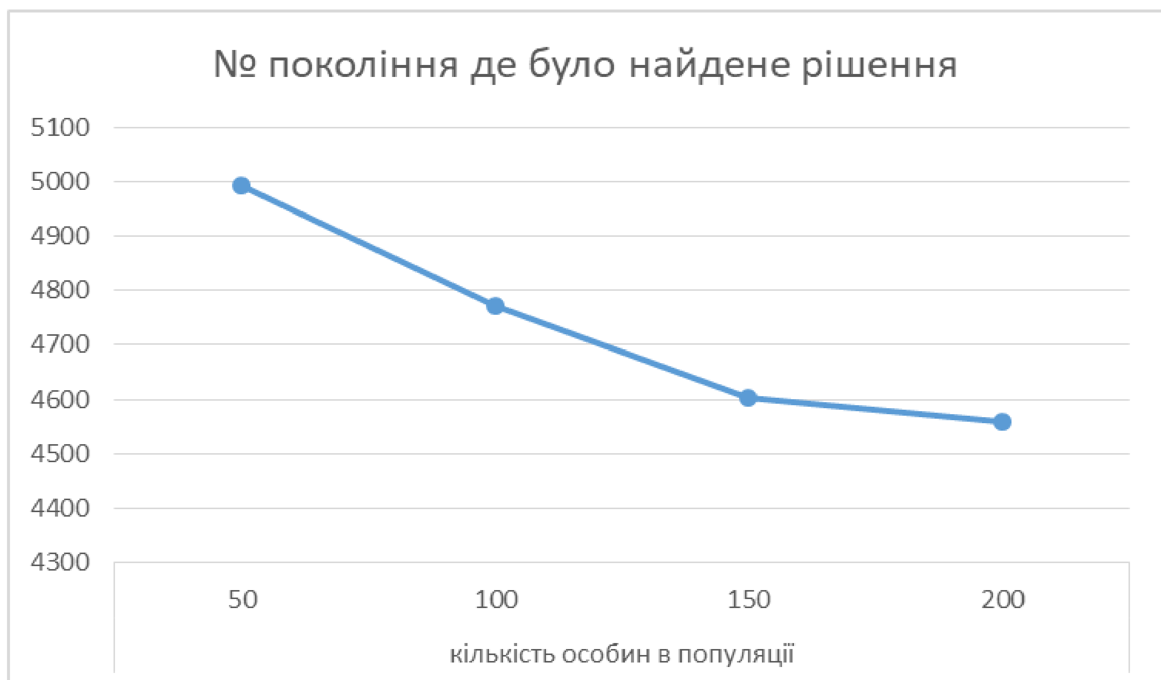


Рис. 3.3. Результати пошуку рішення генетичного алгоритму з турніром на чотири особини схрещуванням половинним склеюванням та мутацією 5%.

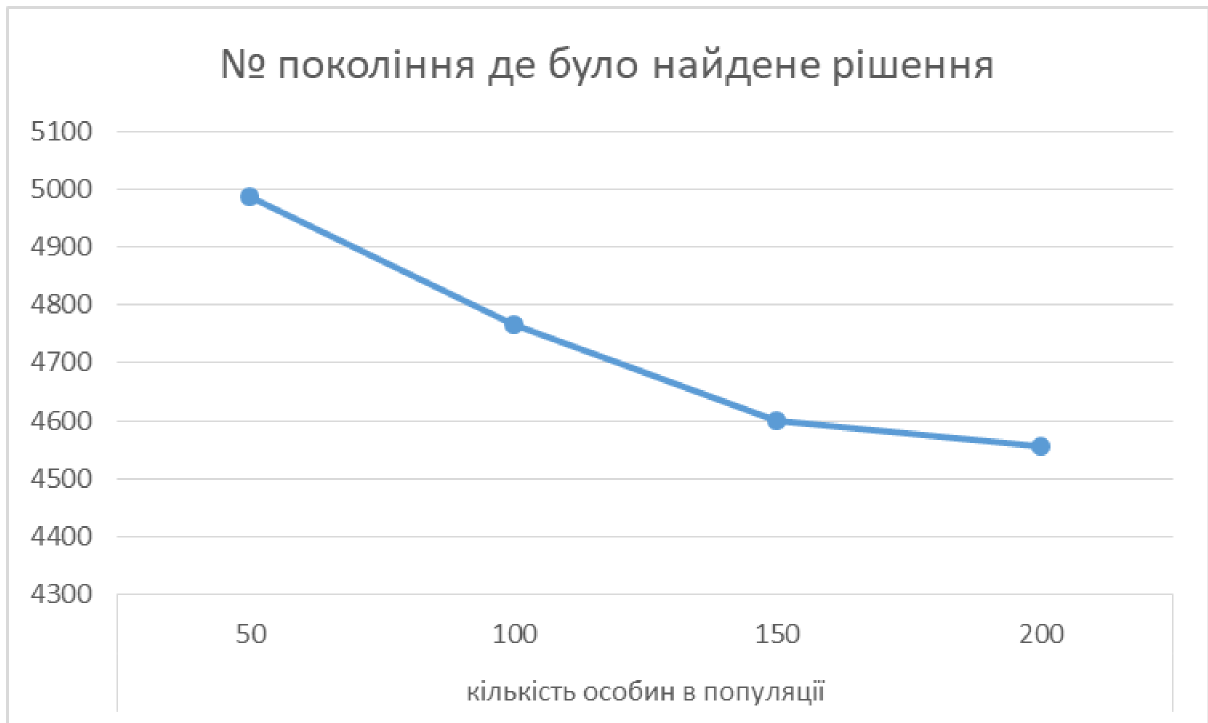


Рис. 3.4. Результати пошуку рішення генетичного алгоритму з турніром на п’ять особин схрещуванням половинним склеюванням та мутацією 5%.

З наведених графіків видно, що суттєве скорочення відбувається при збільшенні кількості учасників з двох до трьох. На наступному кроці збільшення приріст алгоритму суттєво падає, а на останньому кроці збільшення майже зникає. Таким чином достатньо зробити турнір на чотири особини.

З усіх способів схрещення в цій роботі були використані дві найпростіші половинним: склеюванням та гребінкою. Їхні результати за умови турнірного відбору на чотири особини, наведені на рисунках 3.3 та 3.5 відповідно.

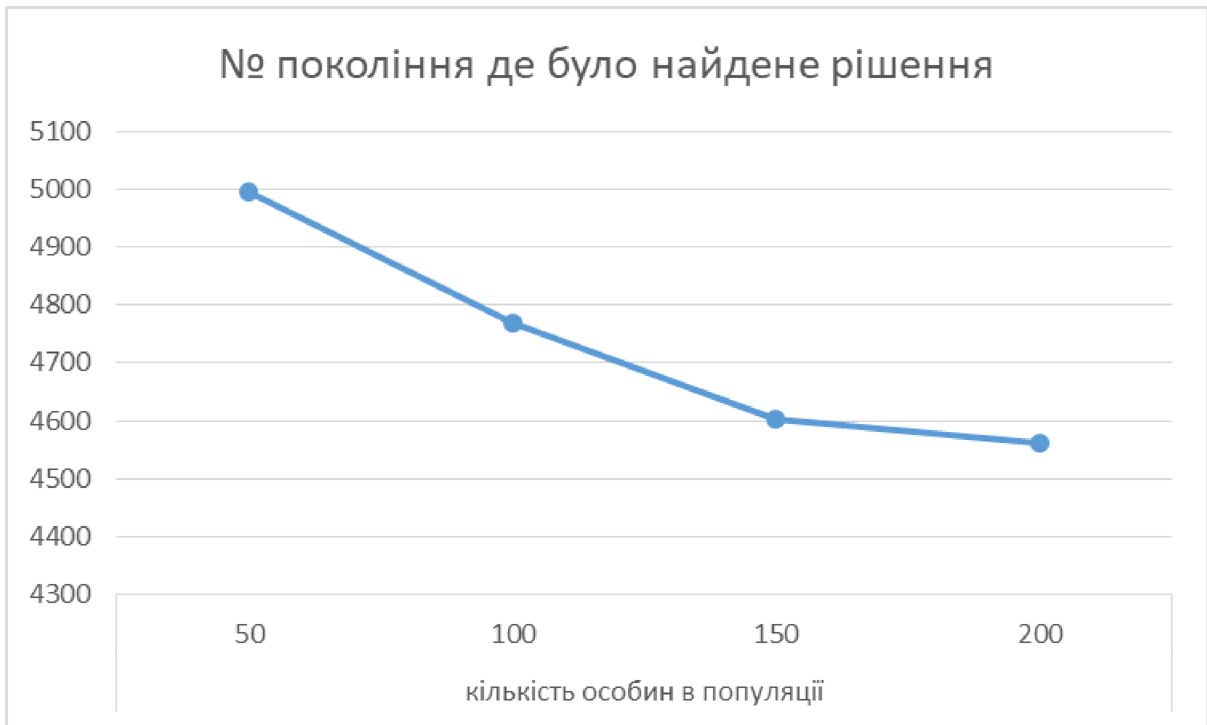


Рис. 3.5. Результати пошуку рішення генетичного алгоритму з турніром на чотири особини схрещуванням гребінкою та мутацією 5%.

Як видно з наведених рисунків, для цього набору даних вибір між цими двома методами схрещування майже не впливає на швидкість збіжності алгоритму, тому надалі використовувались турнірний відбір на чотири особини та схрещуванням половинним склеюванням.

На останньому етапі були виконані дослідження впливу кількості мутованих генів на швидкість збіжності алгоритму. Занадто високий ступінь мутації може призводити до погіршення часових характеристик роботи алгоритму, оскільки алгоритм може проходити повз локальні мінімуми. Результати випробувань для п’яти %, трьох % та одного % мутацій наведені на рисунках 3.3, 3.6 та 3.7 відповідно.

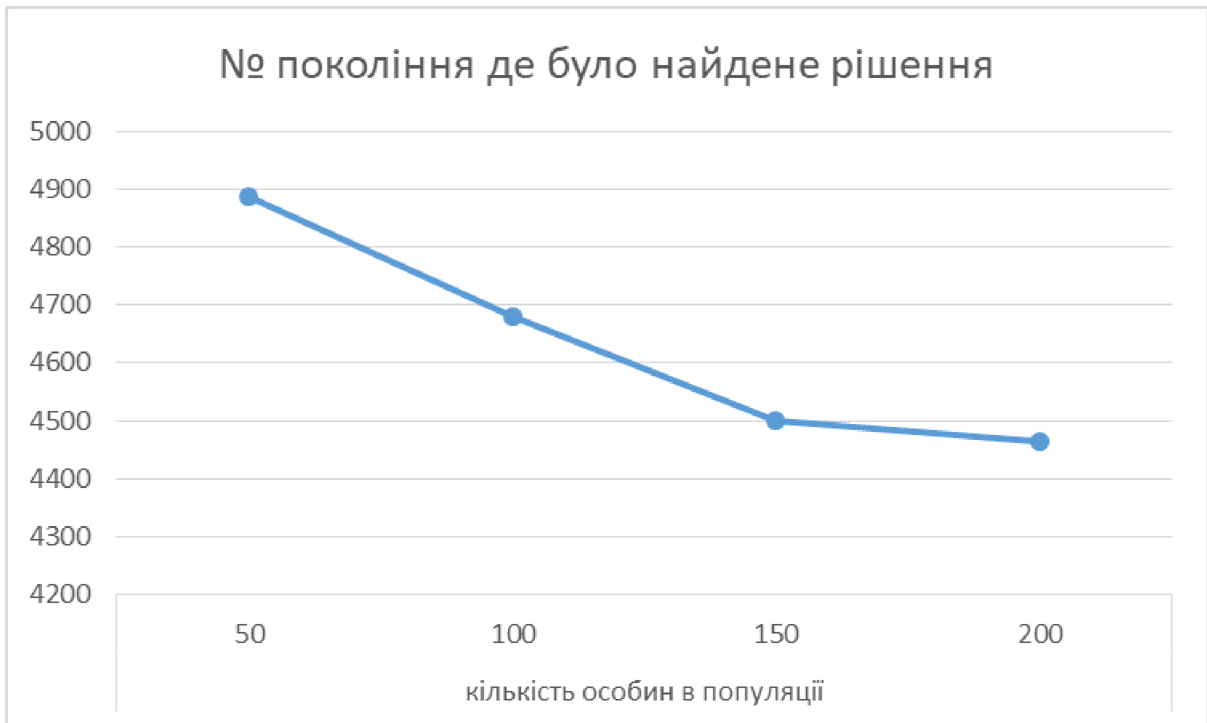


Рис. 3.6. Результати пошуку рішення генетичного алгоритму з турніром на чотири особини схрещуванням половинним склеюванням та мутацією 3%.

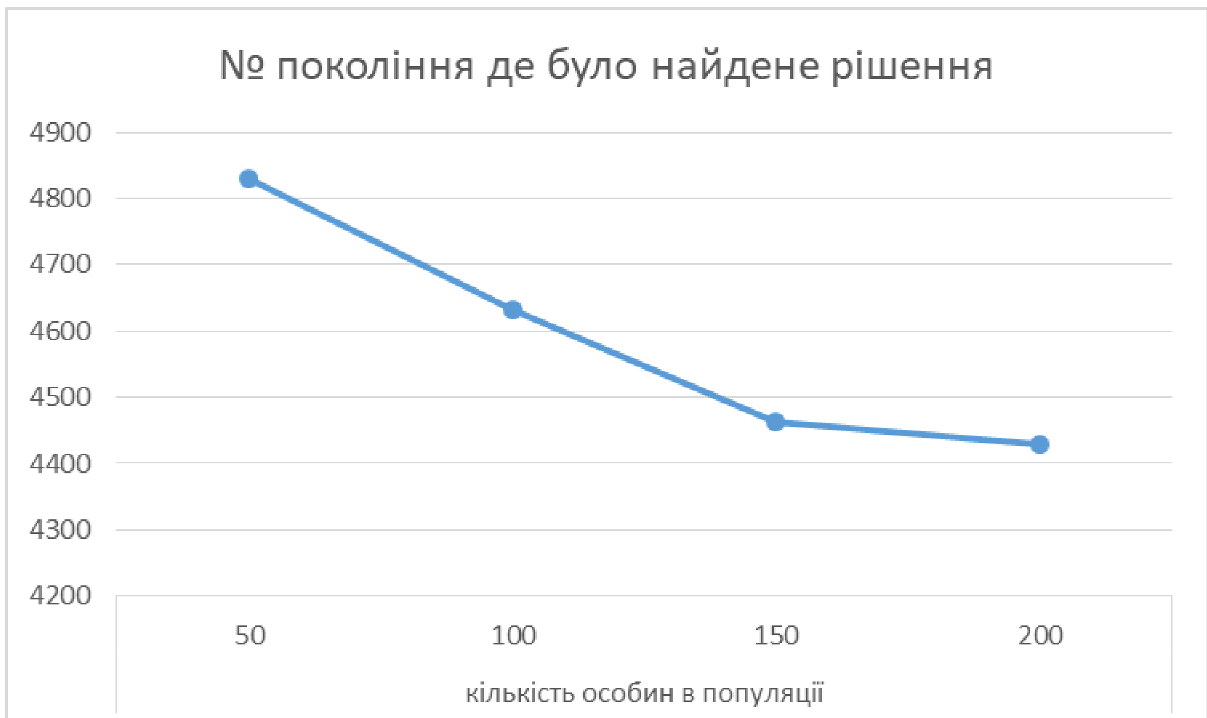


Рис. 3.7. Результати пошуку рішення генетичного алгоритму з турніром на чотири особини схрещуванням половинним склеюванням та мутацією 1%.

3.5. Висновки

В результаті дослідження складання розкладу навчального закладу за допомогою генетичного алгоритму було з’ясовано, що за наявності вхідних даних у вигляді навчального плану, списку студентських груп, навчального навантаження, відомостей про аудиторний фонд та переліку обмежень, еволюційний алгоритм знаходить прийнятне рішення не менш як за чотири тисячі поколінь. При цьому метод схрещування суттєво не впливає на швидкість збіжності алгоритму, для цих даних найкращою кількістю особин в турнірі є чотири, та найкращий ступінь мутації на кожному кроці еволюції є 1%.

ВИСНОВКИ

У кваліфікаційній роботі були розглянуті та порівняні між собою засоби розв’язання NP-повних задач з метою пошуку локального мінімуму. Було наведено визначення цільової функції для планування розкладу. Був обраний генетичний алгоритм як найбільш раціональний спосіб побудови розкладу занять університету за наявними даними з реальної організації навчального процесу, за умови задовільної якості та часу розв’язання задачі.

У другому розділі була представлена реалізація стандартного генетичного алгоритму для вирішення проблеми планування занять, та запропоновано модифікації для покращення продуктивність планування. Ці модифікації базуються на об’єднанні евристичних принципів із принципами GA. Особливістю описаного підходу є використання хромосомного представлення для розкладу занять.

Обробка обмежень у запропонованій методиці відбувається шляхом забезпечення напрямку пошуку до можливого регіону і використання схеми нішування для підтримки різноманітності, що допомагає оператору кросинговеру GA знаходити все кращі рішення з популяції. Це стало можливим завдяки популяційному підходу GA та можливості попарного порівняння рішень за допомогою оператора турнірного відбору.

В результаті дослідження складання розкладу університету за допомогою генетичного алгоритму було з’ясовано, що за наявності вхідних даних у вигляді навчального плану, списку студентських груп, навчального навантаження, відомостей про аудиторний фонд та переліку обмежень, еволюційний алгоритм знаходить прийнятне рішення не менш як за чотири тисячі поколінь. При цьому метод схрещування суттєво не впливає на швидкість збіжності алгоритму, для цих даних найкращою кількістю особин в турнірі є чотири, та найкращий ступінь мутації на кожному кроці еволюції є 1%.

У майбутньому можлива модифікація локальної оптимізації алгоритму, за рахунок використання м’яких обмежень розкладу для підвищення ефективності розв’язання задачі.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dahiya S. Course scheduling with preference optimization - The Pennsylvania State University, 2015. – 68 p.

2. Burke E. University timetabling / E. Burke, S. Petrovic // Handbook of Scheduling: Algorithms, Models, and Performance Analysis. – Chapman Hall : CRC Press, 2004. – Part VI. – Chapter 45. – p. 1–14.

3. Wren A. Scheduling, timetabling and rostering – a special relationship? // Practice and Theory of Automated Timetabling, In E. Burke and P. Ross (editors). - Springer-Verlag LNCS 1153, 1996. - pp. 46–75.

4. Лагоша Б. А. Комплекс моделей и методов оптимизации расписания занятий в вузе / Б. А. Лагоша, А. В. Петропавловская. – М. : Экономика и математические методы. – 1993 г. – 410 с.

5. Попов Г. А. Формализация задачи составления расписания в высшем учебном заведении / Г.А. Попов –Вестник АЕТУ. – 2006. – № 1.

6. Яндыбаева Н. В. Генетический алгоритм в задаче оптимизации учебного расписания вуза / Н. В. Яндыбаева. // Современные наукоемкие технологии. 2009. No 11. с. 97–98.

7. Rudova H. Multi-criteria soft constraints in timetabling / H. Rudova, M. Vlk // Proc. MISTA. – 2005. – p. 11–15.

8. Burke E. Applications to timetabling / E. Burke, J. Kingston, D. de Werra // The Handbook of Graph Theory, Chapman Hall/CRC Press, 2004. – pp. 445–474.

9. Burke E. Automated University Timetabling: The State of the Art / E. Burke, J. Kingston, K. Jackson, R. Weare // The Computer Journal 40(9), 1997. - p. 565–571.

10. Lach G. Curriculum based course timetabling: new solutions to Udine benchmark instances / G. Lach, M.E. Lübbecke // Annals OR 194(1), 2012. - p. 255–272.

11. Shraerf A. A survey of automated timetabling / Journal Artificial Intelligence Review archive, Volume 13, Issue 2, 1999.

12. Yazdani M. Algorithms for university course scheduling problems / M. Yazdani, B. Naderi, E. Zeinali // Tehnički vjesnik 24, Suppl. 2, 2017. - p. 241-247

13. Turabieh H. An integrated hybrid approach to the examination time tabling problem. / H. Turabieh, S. Abdullah // Omega, 39, 2011. - pp. 598–607.

14. Burke E. K. A graph-based hyper-heuristic for educational timetabling problems. / E. K. Burk, B. McCollum, A. Meisels, S. Petrovic, R. Qu // European Journal of Operational Research, 176, 2007. - pp. 177–192.

15. Causmaecker P. D. A decomposed metaheuristic approach for a real-world university timetabling problem. / P. D. Causmaecker, P. Demeester, B.G. Vanden. // European Journal of Operational Research, 195, 2009. - pp. 307–318.

16. Bardadym, V. A. Computer-aided school and university timetabling: The new wave. // Berlin: Springer: Practice and theory of automated timetabling. Lecture notes in computer science, in E. Burke and P. Ross (Eds.), 1153, 1996. - pp. 22–45.

17. Shiau D. F. A hybrid particle swarm optimization for a university course scheduling problem with flexible preferences. // Expert Systems with Applications, 38, 2011. - pp. 235–248.

18. Al-Yakoob S. M. A mixed-integer programming approach to a class timetabling problem: A case study with gender policies and traffic considerations. / S. M. Al-Yakoob, H. D. Sherali // European Journal of Operational Research, 180, 2007. - pp. 1028–1044.

19. Lü Z. Adaptive Tabu Search for course timetabling. / Z. Lü, J. K. Hao // European Journal of Operational Research, 200, 2010. - pp. 235–244.

20. Zhang, D. A simulated annealing with a new neighbourhood structure based algorithm for high school timetabling problems. / D. Zhang, Y. Liu, R. M'Hallah, S. C. H. Leung // European Journal of Operational Research, 203, 2010. - pp. 550–558.

21. Wang Y. Z. An application of genetic algorithm methods for teacher assignment problems. // Expert Systems with Applications, 22, 2002. - pp. 295–302.

22. MirHassani S. A. A computational approach to enhancing course timetabling with integer programming. // Applied Mathematics and Computation, 175, 2006. - pp. 814–822.

23. Burke E. K. Hybrid variable neighbourhood approaches to university exam timetabling / E. K. Burke, A. J. Eckersley, B. McCollum, S. Petrovic, R. Qu // European Journal of Operational Research, 206, 2010. - pp. 46–53.

24. Daskalaki S. Efficient solutions for a university timetabling problem through integer programming. / S. Daskalaki, T. Birbas // European Journal of Operational Research, 160, 2005. - pp. 106–120.

25. Abdullah S. On the use of multi neighbourhood structures within a Tabu-based memetic approach to university timetabling problems. / S. Abdullah, H. Turabieh // Information Sciences, 191, 2012. - pp. 146-168.

26. Thepphakorn T. An ant colony based timetabling tool. / T. Thepphakorn, P. Pongcharoen, C. Hicks // International Journal of Production Economics, 149, 3, 2014. - pp. 131-144.

27. Eiben A.E. Introduction to Evolutionary Computing. / A.E. Eiben, J.E. Smith. - Springer, New York, 2003.

28. Holland J. H. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. - Michigan Press, 1975.

29. Smith K.A. Hopfield neural networks for timetabling: formulations, methods, and comparative results. / K.A. Smith, D. Abramson, D. Duke // Computers & Industrial Engineering, v. 44, 2003. - pp. 283-305.

30. Lim J.H. Timetable scheduling using neural networks with parallel implementation on transputers. / J.H. Lim, K.F. Loe // IEEE Internacional Joint Conference on Neural Networks, v. 1, 1991. - pp. 122-127.

31. Carrasco M.P. A Potts Neural Network Heuristic for the Class/Teacher Timetabling Problem. / M.P. Carrasco, M. Pato // MIC'2001 - 4th Metaheuristics International Conference, Portugal, 2001. - pp. 139-142.

32. Haykin S. Redes Neurais: princípios e prática. 2nd ed. // POA-RS, Bookman, 2001.

33. Arbib M.A. Handbook of brain theory and neural networks. 2nd ed. // Cambridge, MIT Press, 2003.

34. Braga A.P. Redes Neurais Artificiais: teorias e aplicações. / A.P. Braga, A.L.F. Carvalho, T. B. Ludermir // RJ: LTC, 2000.

35. Gislen L. Teachers and Classes with Neural Networks. / L. Gislen, C. Peterson, B. Soderberg // International Journal of Neural Systems, v. 1, 1989. - pp. 167-176.

36. Gianoglio P. Application of neural networks to timetable construction. In: Neuro-Nimes'90 – Proceedings of Third International Workshop on Neural Networks and their Applications, France, 1990. - pp. 53-67.

37. Taborda A. W. M. Neural Networks Applied on Educational Timetabling Problems: an Overview / A. W. M. Taborda, C. A. T. de Carvalho // IIIS, 2009. Access mode: https://www.iiis.org/cds2008/cd2009sci/CCCT2009/PapersPdf/ T810IP.pdf

38. Omara F. A. Genetic Algorithms for Task Scheduling Problem / F. A. Omara, M. M. Arafa // Springer-Verlag Berlin Heidelberg: Foundations of Comput. Intel., Vol. 3, SCI 203, 2009. - pp. 479–507.

39. El-Rewini, H. Task Scheduling in Parallel and Distributed Systems. / H. El-Rewini, T.G. Lewis, H.H .Ali // Prentice-Hall International Editions, 1994.

40. Wu A.S. An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling. / A.S. Wu, H. Yu, S. Jin, K.-C. Lin, G. Schiavone // IEEE Trans. Parallel and Distributed Systems 15, 2004. – pp. 824–834.

41. Kwok Y. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. / Y. Kwok, I. Ahmad // ACM Computing Survey 31, 1999. - pp. 406–471.

42. Bouvry P. Efficient Solutions for Mapping Parallel Programs. / P. Bouvry, J. Chassin, D. Trystram // CWI-Center for Mathematics and computer science, Amsterdam, The Netherlands, 1995.

43. Wu M. Hypertool: A Programming aid for message-passing systems. / M. Wu, D.D. Gajski // IEEE Trans. Parallel Distributed Systems 1, 1990. – pp. 381–422.

44. Levine D. A Parallel Genetic Algorithm for The Set Partitioning Problem, Ph.D. thesis in computer science, Department of Mathematics and computer science, IIlinois Institute of Technology, Chicago, USA, 1994.

45. Back T. Evolutionary Computation: Comments on the History and Current State. / T. Back, U. Hammel, H.-P. Schwefel // IEEE Trans. Evolutionary Computation 1, 1997. – pp. 3–17.

46. Talbi E.G. A new Approach for The Mapping Problem: A Parallel Genetic Algorithm / E.G. Talbi, T. Muntean, // 1993.

47. Ali S. GSA: Scheduling And Allocation Using Genetic Algorithm. / S. Ali, S.M. Sait, M.S.T. Benten // Proceedings of the Conference on EURO-DAC with EURO WDHL, Grenoble, 1994. - pp. 84–89.

48. Hou E.H. A Genetic Algorithm for Multiprocessor Scheduling. / E.H. Hou, N. Ansari, H. Ren // IEEE Trans. Parallel Distributed Systems. 5, 1994. – pp. 113–120.

49. Ahmed I. Task Assignment using a Problem-Space Genetic Algorithm. / I. Ahmed, M.K. Dhodhi // Concurrency. Pract. Exper. 7, 1995. – pp. 411–428.

50. Kwok Y. High performance Algorithms for Compile-time Scheduling of Parallel Processors, Ph.D. Thesis, Hong Kong University, 1997.

51. Tsuchiya T. Genetic-Based Multiprocessor Scheduling Using Task Duplication. / T. Tsuchiya, T. Osada, T. Kikuno // Microprocessors and Microsystems 22, 1998. – pp. 197–207.

52. Blickle T. A Mathematical Analysis of Tournament Selection. / T. Blickle, L. Thiele // In: Proc. of the 6th International Conf. on Genetic Algorithms (ICGA 1995). Morgan Kaufmann, San Francisco, 1995.

53. Kumar S. Efficient Task Mapping on Distributed Heterogeneous Systems for Mesh Applications. / S. Kumar, U. Maulik, S. Bandyopadhyay, S.K. Das // In: Proceedings of the International Workshop on Distributed Computing, Kolkata, India, 2001.

54. Ahmad I. A New Approach to Scheduling Parallel Programs Using Task Duplication. / I. Ahmad, Y. Kwok // In: Proc. of the 23rd International Conf. on Parallel Processing, North Carolina State University, NC, USA, 1994.

55. Ahmad I. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. / I. Ahmad, Y. Kwok // Journal of Parallel and Distributed Computing 95, 1999. – pp. 381– 422.

56. Deb K. An efficient constraint handling method for genetic algorithms, Computer Methods in Applied Mechanics and Engineering, Volume 186, Issues 2–4, 2000. – pp. 311-338,

57. Deb K. Optimization for Engineering Design: Algorithms and Examples. - Prentice-Hall, New Delhi, 1995.

58. Homaifar A. Constrained optimization via genetic algorithms / A. Homaifar, S.H.-V. Lai, X. Qi //Simulation 62 (4), 1994. - pp. 242-254.

59. Michalewicz Z. Evolutionary optimization of constrained problems / Z. Michalewicz, N. Attia // in: A.V. Sebald, L.J. Fogel (Eds.), Proceedings of the Third Annual Conference on Evolutionary Programming, World Scientific, Singapore, 1994. - pp. 98–108

60. Michalewicz Z. Genetic algorithms, numerical optimization, and constraints, in: L. Eshelman (Ed.), Proceedings of the Sixth International Conference on Genetic Algorithms, Morgan Kauffman, San Mateo, 1995. - pp. 151–158.

61. Michalewicz Z. Evolutionary algorithms for constrained parameter optimization problems / Z. Michalewicz, M. Schoenauer // Evolutionary Computation, 4 (1), 1996. - pp. 1-32.

62. Powell D. Using genetic algorithms in engineering design optimization with nonlinear constraints / D. Powell, M.M. Skolnick // in: S. Forrest (Ed.), Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kauffman, San Mateo, 1993. - pp. 424–430.

63. Kim J-H. Evolutionary programming techniques for constraint optimization problems / J-H. Kim, H. Myung // IEEE Transcations on Evolutionary Computation, 1 (2), 1997. - pp. 129-140.

64. Goldberg D. E. Genetic Algorithms, in Search, Optimization & Machine Learning - PHI Fourth Indian Reprint, 2001.

65. Richardson J.T. Some guidelines for genetic algorithms with penalty functions / J.T. Richardson, M.R. Palmer, G. Liepins, M. Hilliard // in: J.D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kauffman, San Mateo, 1989. - pp. 191–197.

66. Deb K. An investigation of niche and species formation in genetic function optimization / K. Deb, D.E. Goldberg // in: J.D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kauffman, San mateo, 1989, - pp. 42–50.

67. Goldberg D.E. Genetic Algorithms in Search, Optimization, and Machine Learning. - Addison-Wesley, Reading, MA, 1989.

68. Методичні рекомендації до виконання кваліфікаційних робіт здобувачами другого (магістерського) рівня вищої освіти спеціальностей 121 «Інженерія програмного забезпечення» та 122 «Комп’ютерні науки» / Б.І. Мороз, О.В. Іванченко, О.В. Реута, О.С. Шевцова; М-во освіти і науки України, Нац. техн. ун-т “Дніпровська політехніка”. – Дніпро : НТУ «ДП», 2021. – 56 с.

Додаток А

ЛІСТИНГ ПРОГРАМИ

Amga2.py

```
from Schedule import Schedule
import functools
import random
from collections import deque
from random import randrange
from time import time

class Amga2:

    def initAlgorithm(self, prototype, numberOfChromosomes=100, replaceByGeneration=8):
        self._prototype = prototype
        if numberOfChromosomes < 2:
            numberOfChromosomes = 2
        self._archivePopulation = []
        self._parentPopulation = []
        self._offspringPopulation = []
        self._combinedPopulation = []
        self._populationSize = self._archiveSize = numberOfChromosomes

    def __init__(self, configuration, etaCross=0.35, mutationSize=2,
crossoverProbability=80,
                 mutationProbability=3):
        self.initAlgorithm(Schedule(configuration))
        self._mutationSize, self._etaCross = mutationSize, etaCross
        self._crossoverProbability, self._mutationProbability = crossoverProbability,
mutationProbability

    @functools.total_ordering
    class DistanceMatrix:

        def __init__(self):
            self.index1 = -1
            self.index2 = -1
            self.distance = 0.0

        def __lt__(self, other):
            if self is None:
                return 0

            if self.distance < other.distance:
                return -1
            if self.distance > other.distance:
                return 1
            if self.index1 < other.index1:
                return -1
            if self.index1 > other.index1:
                return 1
            if self.index2 < other.index2:
                return -1
            if self.index2 > other.index2:
                return 1
            return 0

        def __eq__(self, other):
            return self.index1 == other.index1 and self.index2 == other.index2 and
self.distance == other.distance

        def __ne__(self, other):
            return not self.__eq__(other)

    @property
    def result(self):
        return self._combinedPopulation[0]

    def initialize(self):
        prototype = self._prototype
        archiveSize, populationSize = self._archiveSize, self._populationSize
        archivePopulation = self._archivePopulation = []
        parentPopulation = self._parentPopulation = []
```

```
        offspringPopulation = self._offspringPopulation = []
        combinedPopulation = self._combinedPopulation = []
        for i in range(archiveSize):
            archivePopulation.append(prototype.makeNewFromPrototype())
            combinedPopulation.append(prototype.makeNewFromPrototype())
        for i in range(populationSize):
            parentPopulation.append(prototype.makeNewFromPrototype())
            offspringPopulation.append(prototype.makeNewFromPrototype())
            combinedPopulation.append(prototype.makeNewFromPrototype())

    def assignInfiniteDiversity(self, population, elite):
        for index in elite:
            population[index].diversity = float("inf")

    def assignDiversityMetric(self, population, elite):
        if len(elite) <= 2:
            self.assignInfiniteDiversity(population, elite)
            return
        distinct = self.extractDistinctIndividuals(population, elite)
        if len(distinct) <= 2:
            self.assignInfiniteDiversity(population, elite)
            return

        size = len(distinct)
        for e in distinct:
            population[e].diversity = 0.0
        val = population[distinct[size - 1]].getDifference(population[distinct[0]])
        if val == 0:
            return
        for j in range(size):
            if j == 0:
                diff = population[distinct[j + 1]].getDifference(population[distinct[j]])
                hashArray = (0.0, population[distinct[j]].fitness, population[distinct[j + 
1]].fitness)
                r = diff / val
                population[distinct[j]].diversity += (r * r)
            elif j == size - 1:
                diff = population[distinct[j]].getDifference(population[distinct[j - 1]])
                l = diff / val
                population[distinct[j]].diversity += (l * l)
            else:
                diff = population[distinct[j]].getDifference(population[distinct[j - 1]])
                l = diff / val
                diff = population[distinct[j + 1]].getDifference(population[distinct[j]])
                r = diff / val
                population[distinct[j]].diversity += (l * r)

    def createOffspringPopulation(self):
        currentArchiveSize, populationSize = self._currentArchiveSize, self._populationSize
        archivePopulation, parentPopulation, offspringPopulation = self._archivePopulation, 
self._parentPopulation,self._offspringPopulation
        etaCross, crossoverProbability = self._etaCross, self._crossoverProbability
        for i in range(populationSize):
            r1 = -1
            while r1 < 0 or archivePopulation[r1] == archivePopulation[i]:
                r1 = randrange(currentArchiveSize)
            r2 = -1
            while r2 < 0 or archivePopulation[r2] == archivePopulation[i] or r2 == r1:
                r2 = randrange(currentArchiveSize)
            r3 = -1
            while r3 < 0 or archivePopulation[r3] == archivePopulation[i] or r3 == r1 or r3 
== r2:
                r3 = randrange(currentArchiveSize)
            offspringPopulation[i] = offspringPopulation[i].crossovers(parentPopulation[i], 
archivePopulation[r1],

archivePopulation[r2], archivePopulation[r3],
                                                                           etaCross, 
crossoverProbability)
            offspringPopulation[i].rank = parentPopulation[i].rank

    @functools.total_ordering
    def checkDomination(self, a, b):
        if a.fitness < b.fitness:
            return -1
        if a.fitness > b.fitness:
            return 1
        return 0
```

```
    def extractDistinctIndividuals(self, population, elite):
        return sorted(set(elite), key=lambda e: population[e].fitness)

    def extractENNSPopulation(self, mixedPopulation, pool, desiredEliteSize):
        poolSize, mixedSize = len(pool), len(mixedPopulation)
        filtered = [index for index in pool if mixedPopulation[index].diversity ==
float("inf")]
        numInf = len(filtered)
        if desiredEliteSize <= numInf:
            return filtered[:desiredEliteSize]
        elite = deque(dict.fromkeys(pool))
        pool.clear()
        if desiredEliteSize <= numInf:
            return elite
        distance = [[0 for x in range(poolSize)] for y in range(poolSize)]
        indexArray = poolSize * [0]
        originalArray = mixedSize * [-1]
        for counter, index in enumerate(elite):
            indexArray[counter] = index
            originalArray[index] = counter
        distArray = []
        for i in range(poolSize):
            for j in range(i + 1, poolSize):
                distMatrix = Amga2.DistanceMatrix()
                distMatrix.index1 = indexArray[i]
                distMatrix.index2 = indexArray[j]
                distMatrix.distance =
mixedPopulation[distMatrix.index1].getDifference(mixedPopulation[distMatrix.index2])
                distance[j][i] = distance[i][j] = distMatrix.distance
                distArray.append(distMatrix)
        distArray.sort()
        distArray_len = len(distArray)
        idx = 0
        while len(elite) > desiredEliteSize and idx < distArray_len:
            temp = distArray[idx]
            idx += 1
            index1, index2 = temp.index1, temp.index2
            while (originalArray[index1] == -1 or originalArray[index2] == -1) and idx <
distArray_len:
                temp = distArray[idx]
                idx += 1
                index1, index2 = temp.index1, temp.index2
            if idx >= distArray_len:
                break
            if mixedPopulation[index1].diversity == float("inf") and
mixedPopulation[index2].diversity == float("inf"):
                continue
            if mixedPopulation[index1].diversity == float("inf"):
                elite.remove(index2)
                pool.append(index2)
                originalArray[index2] = -1
            elif mixedPopulation[index2].diversity == float("inf"):
                elite.remove(index1)
                pool.append(index1)
                originalArray[index1] = -1
            else:
                dist1 = float("inf")
                for index in elite:
                    if index != index1 and index != index2:
                        if dist1 > distance[originalArray[index1]][originalArray[index]]:
                            dist1 = distance[originalArray[index1]][originalArray[index]]
                dist2 = float("inf")
                for index in elite:
                    if index != index1 and index != index2:
                        if dist2 > distance[originalArray[index2]][originalArray[index]]:
                            dist2 = distance[originalArray[index2]][originalArray[index]]
                if dist1 < dist2:
                    elite.remove(index1)
                    pool.append(index1)
                    originalArray[index1] = -1
                else:
                    elite.remove(index2)
                    pool.append(index2)
                    originalArray[index2] = -1
        while len(elite) > desiredEliteSize:
            pool.append(elite.popleft())
        return elite

    def extractBestRank(self, population, pool, elite):
```

```
        if not pool:
            return False
        checkDomination = self.checkDomination
        remains = deque()
        elite.append(pool.popleft())
        while pool:
            index1 = pool.popleft()
            flag, index2 = -1, 0
            while index2 < len(elite):
                flag = checkDomination(population[index1], population[index2])
                if flag == 1:
                    remains.append(index2)
                    del elite[index2]
                elif flag == -1:
                    break
                else:
                    index2 += 1
            if flag > -1:
                elite.append(index1)
            else:
                remains.append(index1)
        pool.clear()
        pool.extend(remains)
        return True

    def fillBestPopulation(self, mixedPopulation, mixedLength, population,
populationLength):
        pool = deque(range(mixedLength))
        elite, filled = deque(), []
        rank = 1
        assignInfiniteDiversity = self.assignInfiniteDiversity
        extractBestRank, extractENNSPopulation = self.extractBestRank,
self.extractENNSPopulation
        for index in pool:
            mixedPopulation[index].diversity = 0
        hasBetter = True
        while hasBetter and len(filled) < populationLength:
            hasBetter = extractBestRank(mixedPopulation, pool, elite)
            for index in elite:
                mixedPopulation[index].rank = rank
            if rank == 1:
                assignInfiniteDiversity(mixedPopulation, elite)
            rank += 1
            if len(elite) + len(filled) < populationLength:
                filled.extend(elite)
                elite.clear()
            else:
                temp = extractENNSPopulation(mixedPopulation, elite, populationLength -
len(filled))
                filled.extend(temp)
        for j, index in enumerate(filled):
            population[j] = mixedPopulation[index]

    def fillDiversePopulation(self, mixedPopulation, pool, population, startLocation,
desiredSize):
        self.assignDiversityMetric(mixedPopulation, pool)
        poolSize = len(pool)
        indexArray = sorted(pool, key=lambda e: mixedPopulation[e].diversity)
        for i in range(desiredSize):
            population[startLocation + i] = mixedPopulation[indexArray[poolSize - 1 - i]]

    def createParentPopulation(self):
        pool = deque(range(self._currentArchiveSize))
        elite, selectionPool = [], deque()
        rank, populationSize = 1, self._populationSize
        archivePopulation, parentPopulation = self._archivePopulation,
self._parentPopulation
        extractBestRank = self.extractBestRank
        while len(selectionPool) < populationSize:
            extractBestRank(archivePopulation, pool, elite)
            for i in elite:
                archivePopulation[i].rank = rank
                selectionPool.append(i)
            rank += 1
            elite.clear()
        j = 0
        for i in selectionPool:
            parentPopulation[j] = archivePopulation[i]
            j += 1
```

```
        self.fillDiversePopulation(archivePopulation, selectionPool, parentPopulation, j,
populationSize - j)

    def mutateOffspringPopulation(self):
        currentArchiveSize, populationSize = self._currentArchiveSize, self._populationSize
        mutationProbability, mutationSize = self._mutationProbability, self._mutationSize
        offspringPopulation = self._offspringPopulation
        for i in range(populationSize):
            pMut = mutationProbability + (1.0 - mutationProbability) * (
                        float(offspringPopulation[i].rank - 1) / (currentArchiveSize - 1))
            offspringPopulation[i].mutation(mutationSize, pMut)

    def updateArchivePopulation(self):
        currentArchiveSize, populationSize = self._currentArchiveSize, self._populationSize
        archivePopulation, combinedPopulation, offspringPopulation =
self._archivePopulation, self._combinedPopulation, self._offspringPopulation
        if (currentArchiveSize + populationSize) <= self._archiveSize:
            for j, i in enumerate(range(populationSize), start=currentArchiveSize):
                archivePopulation[j] = offspringPopulation[i]
            self._currentArchiveSize += populationSize
        else:
            for i in range(currentArchiveSize):
                combinedPopulation[i] = archivePopulation[i]
            for i in range(populationSize):
                combinedPopulation[currentArchiveSize + i] = offspringPopulation[i]
            self.fillBestPopulation(combinedPopulation, currentArchiveSize + populationSize,
archivePopulation,
                                    self._archiveSize)
            self._currentArchiveSize = self._archiveSize
        for e in archivePopulation:
            e.diversity = 0

    def finalizePopulation(self):
        currentArchiveSize, populationSize = self._currentArchiveSize, self._populationSize
        archivePopulation, combinedPopulation = self._archivePopulation,
self._combinedPopulation
        elite = []
        pool = deque([i for i in range(currentArchiveSize) if archivePopulation[i].fitness
>= 0])
        if pool:
            self.extractBestRank(archivePopulation, pool, elite)
            pool.clear()
            if len(elite) > populationSize:
                for index in elite:
                    archivePopulation[index].diversity = 0
                self.assignInfiniteDiversity(archivePopulation, elite)
                self.extractENNSPopulation(archivePopulation, pool, populationSize)
                elite = list(pool)
            self._currentArchiveSize = len(elite)
            for i, index in enumerate(elite):
                combinedPopulation[i] = archivePopulation[index]
        else:
            self._currentArchiveSize = 0

    def reform(self):
        random.seed(round(time() * 1000))
        if self._crossoverProbability < 95:
            self._crossoverProbability += 1.0;
        elif self._mutationProbability < 30:
            self._mutationProbability += 1.0;

    def run(self, maxRepeat=9999, minFitness=0.999):
        self.initialize()
        self._currentArchiveSize = self._populationSize
        createParentPopulation, createOffspringPopulation = self.createParentPopulation,
self.createOffspringPopulation
        mutateOffspringPopulation, updateArchivePopulation = self.mutateOffspringPopulation,
self.updateArchivePopulation
        random.seed(round(time() * 1000))
        currentGeneration = 0
        repeat, lastBestFit = 0, 0.0
        while True:
            if currentGeneration > 0:
                bestFitness = self.result.fitness
                print("Fitness:", "{:f}\t".format(bestFitness), "Generation:",
currentGeneration, end="\r")
                if bestFitness > minFitness:
                    self.finalizePopulation()
                    break
```

```
                difference = abs(bestFitness - lastBestFit)
                if difference <= 0.0000001:
                    repeat += 1
                else:
                    repeat = 0
                if repeat > (maxRepeat / 100):
                    self.reform()
                lastBestFit = bestFitness
            createParentPopulation()
            createOffspringPopulation()
            mutateOffspringPopulation()
            updateArchivePopulation()
            currentGeneration += 1

    def __str__(self):
        return "Archive-based Micro Genetic Algorithm (AMGA2)"
```

Emosoa.py

```
from .NsgaII import NsgaII
import numpy as np
from time import time

class Emosoa(NsgaII):

    def __init__(self, configuration, numberOfCrossoverPoints=2, mutationSize=2,
crossoverProbability=80,
                 mutationProbability=3, maxIterations=5000):
        self._max_iterations = maxIterations
        super().__init__(configuration, numberOfCrossoverPoints, mutationSize,
crossoverProbability,
                        mutationProbability)
        self._currentGeneration, self._repeatRatio = 0, 0
        self._bestScore, self._gBestScore = [], 0
        self._current_position, self._gBest = [], []

    def exploitation(self):
        positions = self._current_position
        dim = positions.shape
        tau = 2 * np.pi
        A = 2 - self._currentGeneration * (2 / self._max_iterations)
        B = (2 * A * A) * np.random.random(dim)
        C = A * positions
        M = B * (self._gBest - positions)
        D, theta = np.abs(C + M), np.random.uniform(0, tau, dim)
        r = np.exp(theta)
        x, y, z = r * np.cos(theta), r * np.sin(theta), r * theta
        positions = D * x * y * z + self._gBest

    def replacement(self, population):
        populationSize = len(population)
        climax = .9
        for i in range(populationSize):
            fitness = population[i].fitness
            if fitness < self._bestScore[i]:
                population[i].updatePositions(self._current_position[i])
                fitness = population[i].fitness
            if fitness > self._bestScore[i]:
                self._bestScore[i] = fitness
                population[i].extractPositions(self._current_position[i])
            if fitness > self._gBestScore:
                self._gBestScore = fitness
                population[i].extractPositions(self._current_position[i])
                self._gBest = self._current_position[i][:]
            if self._repeatRatio > climax and self._gBestScore > climax:
                if i > (populationSize * self._repeatRatio):
                    population[i].updatePositions(self._current_position[i])
        self.exploitation()
        return super().replacement(population)

    def initialize(self, population):
        prototype = self._prototype
        size = 0
        populationSize = len(population)
        for i in range(populationSize):
            positions = []
            population[i] = searchAgent = prototype.makeNewFromPrototype(positions)
            if i < 1:
                size = len(positions)
```

```
                self._current_position = np.zeros((populationSize, size), dtype=float)
                self._gBest = np.zeros(populationSize, dtype=float)
                self._bestScore = np.zeros(populationSize, dtype=float)
            self._bestScore[i] = searchAgent.fitness
            self._current_position[i] = positions

    def run(self, maxRepeat=9999, minFitness=0.999):
        mutationSize = self._mutationSize
        mutationProbability = self._mutationProbability
        nonDominatedSorting = self.nonDominatedSorting
        selection = self.selection
        populationSize = self._populationSize
        population = populationSize * [None]
        self.initialize(population)
        np.random.seed(int(time()))
        currentGeneration = self._currentGeneration
        repeat, lastBestFit = 0, 0.0
        while currentGeneration < self._max_iterations:
            if currentGeneration > 0:
                bestFitness = self.result.fitness
                print("Fitness:", "{:f}\t".format(bestFitness), "Generation:",
currentGeneration, end="\r")
                if bestFitness > minFitness:
                    break
                difference = abs(bestFitness - lastBestFit)
                if difference <= 0.0000001:
                    repeat += 1
                else:
                    repeat = 0
                self._repeatRatio = repeat * 100 / maxRepeat
                if repeat > (maxRepeat / 100):
                    self.reform()
            offspring = self.replacement(population)
            for child in offspring:
                child.mutation(mutationSize, mutationProbability)
            totalChromosome = population + offspring
            front = nonDominatedSorting(totalChromosome)
            if len(front) == 0:
                break
            population = selection(front, totalChromosome)
            self._populationSize = populationSize = len(population)
            if currentGeneration == 0:
                self._chromosomes = population
            else:
                totalChromosome = population + self._chromosomes
                newBestFront = nonDominatedSorting(totalChromosome)
                if len(newBestFront) == 0:
                    break
                self._chromosomes = selection(newBestFront, totalChromosome)
                lastBestFit = bestFitness
            currentGeneration += 1
            self._currentGeneration = currentGeneration

    def __str__(self):
        return "Evolutionary multi-objective seagull optimization algorithm for global
optimization (EMoSOA)"
```

GeneticAlgorithm.py

```
from Schedule import Schedule
import random
from random import randrange
from time import time

class GeneticAlgorithm:

    def initAlgorithm(self, prototype, numberOfChromosomes=100, replaceByGeneration=8,
trackBest=5):
        self._currentBestSize = 0
        self._prototype = prototype
        if numberOfChromosomes < 2:
            numberOfChromosomes = 2
        if trackBest < 1:
            trackBest = 1
        self._chromosomes = numberOfChromosomes * [None]
        self._bestFlags = numberOfChromosomes * [False]
        self._bestChromosomes = trackBest * [0]
        self.set_replace_by_generation(replaceByGeneration)
```

```
    def __init__(self, configuration, numberOfCrossoverPoints=2, mutationSize=2,
crossoverProbability=80,
                 mutationProbability=3):
        self.initAlgorithm(Schedule(configuration))
        self._mutationSize = mutationSize
        self._numberOfCrossoverPoints = numberOfCrossoverPoints
        self._crossoverProbability = crossoverProbability
        self._mutationProbability = mutationProbability

    @property
    def result(self):
        return self._chromosomes[self._bestChromosomes[0]]

    def set_replace_by_generation(self, value):
        numberOfChromosomes = len(self._chromosomes)
        trackBest = len(self._bestChromosomes)
        if (value > numberOfChromosomes - trackBest):
            value = numberOfChromosomes - trackBest
        self._replaceByGeneration = value

    def addToBest(self, chromosomeIndex):
        bestChromosomes = self._bestChromosomes
        length_best = len(bestChromosomes)
        bestFlags = self._bestFlags
        chromosomes = self._chromosomes
        if (self._currentBestSize == length_best and
chromosomes[bestChromosomes[self._currentBestSize - 1]].fitness >=
            chromosomes[chromosomeIndex].fitness) or bestFlags[chromosomeIndex]:
            return

        j = self._currentBestSize
        for i in range(j, -1, -1):
            j = i
            pos = bestChromosomes[i - 1]
            if i < length_best:
                if chromosomes[pos].fitness > chromosomes[chromosomeIndex].fitness:
                    break
                bestChromosomes[i] = pos
            else:
                bestFlags[pos] = False
        bestChromosomes[j] = chromosomeIndex
        bestFlags[chromosomeIndex] = True
        if self._currentBestSize < length_best:
            self._currentBestSize += 1

    def isInBest(self, chromosomeIndex) -> bool:
        return self._bestFlags[chromosomeIndex]

    def clearBest(self):
        self._bestFlags = len(self._bestFlags) * [False]
        self._currentBestSize = 0

    def initialize(self, population):
        prototype = self._prototype
        length_chromosomes = len(population)
        for i in range(0, length_chromosomes):
            population[i] = prototype.makeNewFromPrototype()

    def selection(self, population):
        length_chromosomes = len(population)
        return (population[randrange(32768) % length_chromosomes],
population[randrange(32768) % length_chromosomes])

    def replacement(self, population, replaceByGeneration) -> []:
        mutationSize = self._mutationSize
        numberOfCrossoverPoints = self._numberOfCrossoverPoints
        crossoverProbability = self._crossoverProbability
        mutationProbability = self._mutationProbability
        selection = self.selection
        isInBest = self.isInBest
        length_chromosomes = len(population)
        offspring = replaceByGeneration * [None]
        for j in range(replaceByGeneration):
            parent = selection(population)
            offspring[j] = parent[0].crossover(parent[1], numberOfCrossoverPoints,
crossoverProbability)
            offspring[j].mutation(mutationSize, mutationProbability)
            ci = randrange(32768) % length_chromosomes
            while isInBest(ci):
```

```
                ci = randrange(32768) % length_chromosomes
            population[ci] = offspring[j]
            self.addToBest(ci)
        return offspring

    def run(self, maxRepeat=9999, minFitness=0.999):
        self.clearBest()
        length_chromosomes = len(self._chromosomes)

        self.initialize(self._chromosomes)
        random.seed(round(time() * 1000))
        currentGeneration = 0
        repeat = 0
        lastBestFit = 0.0
        while True:
            best = self.result
            print("Fitness:", "{:f}\t".format(best.fitness), "Generation:",
currentGeneration, end="\r")
            if best.fitness > minFitness:
                break
            difference = abs(best.fitness - lastBestFit)
            if difference <= 0.0000001:
                repeat += 1
            else:
                repeat = 0
            if repeat > (maxRepeat / 100):
                random.seed(round(time() * 1000))
                self.set_replace_by_generation(self._replaceByGeneration * 3)
                self._crossoverProbability += 1
            self.replacement(self._chromosomes, self._replaceByGeneration)
            lastBestFit = best.fitness
            currentGeneration += 1

    def __str__(self):
        return "Genetic Algorithm"
```

Hgasso.py

```
from .NsgaII import NsgaII
import numpy as np

class Hgasso(NsgaII):

    def __init__(self, configuration, numberOfCrossoverPoints=2, mutationSize=2,
crossoverProbability=80,
                 mutationProbability=3):
        super().__init__(configuration, numberOfCrossoverPoints, mutationSize,
crossoverProbability,
                        mutationProbability)
        self._decline = .25
        self._sBestScore, self._sgBestScore = [], 0
        self._sBest, self._sgBest = [], []
        self._current_position, self._velocity = [], []
        self._motility = []

    def replacement(self, population):
        populationSize = len(population)
        climax, decline = 1 - self._decline, self._decline
        for i in range(populationSize):
            fitness = population[i].fitness
            if fitness < self._sBestScore[i]:
                population[i].updatePositions(self._current_position[i])
                fitness = population[i].fitness
                self._motility[i] = True
            if fitness > self._sBestScore[i]:
                self._sBestScore[i] = fitness
                population[i].extractPositions(self._current_position[i])
                self._sBest[i] = self._current_position[i][:]
            if fitness > self._sgBestScore:
                self._sgBestScore = fitness
                population[i].extractPositions(self._current_position[i])
                self._sgBest = self._current_position[i][:]
            if self._repeatRatio > self._sBestScore[i]:
                self._sBestScore[i] -= self._repeatRatio * decline
            if self._repeatRatio > climax and self._sgBestScore > climax:
                if i > (populationSize * self._sgBestScore):
                    population[i].updatePositions(self._current_position[i])
                    self._motility[i] = True
        self.updateVelocities(population)
```

```
        return super().replacement(population)

    def initialize(self, population):
        prototype = self._prototype
        size = 0
        populationSize = len(population)
        for i in range(populationSize):
            positions = []
            population[i] = prototype.makeNewFromPrototype(positions)
            if i < 1:
                size = len(positions)
                self._current_position = np.zeros((populationSize, size), dtype=float)
                self._velocity = np.zeros((populationSize, size), dtype=float)
                self._sBest = np.zeros((populationSize, size), dtype=float)
                self._sgBest = np.zeros(populationSize, dtype=float)
                self._sBestScore = np.zeros(populationSize, dtype=float)
                self._motility = np.zeros(populationSize, dtype=bool)
            self._sBestScore[i] = population[i].fitness
            self._current_position[i] = positions
            self._velocity[i] = np.random.uniform(-.6464, .7157, size) / 3.0

    def updateVelocities(self, population):
        motility = self._motility
        if np.count_nonzero(motility) < 1:
            return
        populationSize = len(population)
        dim = self._velocity[motility].shape
        self._velocity[motility] = np.random.random(dim) * np.log10(np.random.uniform(7.0,
14.0, dim)) * self._velocity[motility] + \
        np.log10(np.random.uniform(7.0, 14.0, dim)) * np.log10(np.random.uniform(35.5, 38.5,
dim)) * (self._sBest[motility] - self._current_position[motility]) + \
        np.log10(np.random.uniform(7.0, 14.0, dim)) * np.log10(np.random.uniform(35.5, 38.5,
dim)) * (self._sgBest - self._current_position[motility])
        self._current_position[motility] += self._velocity[motility]

    def __str__(self):
        return "Hybrid Genetic Algorithm and Sperm Swarm Optimization (HGASSO)"
```

Ngra.py

```
from .NsgaII import NsgaII
from random import random

class Ngra(NsgaII):

    def __init__(self, configuration, numberOfCrossoverPoints=2, mutationSize=2,
crossoverProbability=80,
                 mutationProbability=3):
        NsgaII.__init__(self, configuration, numberOfCrossoverPoints, mutationSize,
crossoverProbability, mutationProbability)

    @staticmethod
    def __cumulative(lists):
        cu_list = []
        length = len(lists)
        cu_list = [sum(lists[0:x:1]) for x in range(0, length+1)]
        return cu_list[1:]

    def replacement(self, population):
        populationSize = self._populationSize
        numberOfCrossoverPoints = self._numberOfCrossoverPoints
        crossoverProbability = self._crossoverProbability
        obj = {m: population[m].fitness for m in range(populationSize)}
        sortedIndices = list(reversed(sorted(obj, key=obj.get)))
        totalFitness = (populationSize + 1) * populationSize / 2
        probSelection = [i / totalFitness for i in range(populationSize)]
        cumProb = self.__cumulative(probSelection)
        selectIndices = [random() for i in range(populationSize)]
        parent = 2 * [None]
        parentIndex = 0
        offspring = []
        for i in range(populationSize):
            selected = False
            for j in range(populationSize - 1):
                if cumProb[j] < selectIndices[i] and cumProb[j + 1] >= selectIndices[i]:
                    parent[parentIndex % 2] = population[sortedIndices[j + 1]]
                    parentIndex += 1
                    selected = True
                    break
```

```
            if not selected:
                parent[parentIndex % 2] = population[sortedIndices[i]]
                parentIndex += 1
            if parentIndex % 2 == 0:
                child0 = parent[0].crossover(parent[1], numberOfCrossoverPoints, 
crossoverProbability)
                child1 = parent[1].crossover(parent[0], numberOfCrossoverPoints, 
crossoverProbability)
                offspring.extend((child0, child1))
        return offspring

    def initialize(self, population):
        super().initialize(population)
        offspring = self.replacement(population)
        population.clear()
        population.extend(offspring)

    def __str__(self):
        return "Non-dominated Ranking Genetic Algorithm (NRGA)"
```

NsgaII.py

```
from Schedule import Schedule
import numpy as np
import random
import sys
from time import time

class NsgaII:

    def initAlgorithm(self, prototype, numberOfChromosomes=100, replaceByGeneration=8):
        self._prototype = prototype
        if numberOfChromosomes < 2:
            numberOfChromosomes = 2
        self._chromosomes = []
        self._populationSize = numberOfChromosomes
        self._repeatRatio = .0

    def __init__(self, configuration, numberOfCrossoverPoints=2, mutationSize=2, 
crossoverProbability=80,
                 mutationProbability=3):
        self.initAlgorithm(Schedule(configuration))
        self._mutationSize = mutationSize
        self._numberOfCrossoverPoints = numberOfCrossoverPoints
        self._crossoverProbability = crossoverProbability
        self._mutationProbability = mutationProbability

    @property
    def result(self):
        return self._chromosomes[0]

    def nonDominatedSorting(self, totalChromosome):
        doublePopulationSize = self._populationSize * 2
        s = doublePopulationSize * [ set() ]
        n = np.zeros(doublePopulationSize, dtype=int)
        front = [ set() ]
        for p in range(doublePopulationSize):
            for q in range(doublePopulationSize):
                if totalChromosome[p].fitness > totalChromosome[q].fitness:
                    s[p].add(q)
                elif totalChromosome[p].fitness < totalChromosome[q].fitness:
                    n[p] += 1
            if n[p] == 0:
                front[0].add(p)
        i = 0
        while front[i]:
            Q = set()
            for p in front[i]:
                for q in s[p]:
                    n[q] -= 1
                    if n[q] == 0:
                        Q.add(q)
            i += 1
            front.append(Q)
        front.pop()
        return front

    def calculateCrowdingDistance(self, front, totalChromosome):
        distance, obj = {}, {}
```

```
        for key in front:
            distance[key] = 0
            fitness = totalChromosome[key].fitness
            if fitness not in obj.values():
                obj[key] = fitness
        sorted_keys = sorted(obj, key=obj.get)
        size = len(obj)
        distance[sorted_keys[0]] = distance[sorted_keys[-1]] = sys.float_info.max
        if size > 1:
            diff2 = totalChromosome[sorted_keys[-
1]].getDifference(totalChromosome[sorted_keys[0]])
            for i in range(1, size - 1):
                diff = totalChromosome[sorted_keys[i +
1]].getDifference(totalChromosome[sorted_keys[i - 1]]) / diff2
                distance[sorted_keys[i]] += diff
        return distance

    def selection(self, front, totalChromosome):
        populationSize = self._populationSize
        calculateCrowdingDistance = self.calculateCrowdingDistance
        N = 0
        newPop = []
        while N < populationSize:
            for row in front:
                N += len(row)
                if N > populationSize:
                    distance = calculateCrowdingDistance(row, totalChromosome)
                    sortedCdf = sorted(distance, key=distance.get, reverse=True)
                    for j in sortedCdf:
                        if len(newPop) >= populationSize:
                            break
                        newPop.append(j)
                    break
                newPop.extend(row)
        return [totalChromosome[n] for n in newPop]

    def replacement(self, population):
        populationSize = self._populationSize
        numberOfCrossoverPoints = self._numberOfCrossoverPoints
        crossoverProbability = self._crossoverProbability
        offspring = []
        S = np.arange(populationSize)
        np.random.shuffle(S)
        halfPopulationSize = populationSize // 2
        for m in range(halfPopulationSize):
            parent0 = population[S[2 * m]]
            parent1 = population[S[2 * m + 1]]
            child0 = parent0.crossover(parent1, numberOfCrossoverPoints,
crossoverProbability)
            child1 = parent1.crossover(parent0, numberOfCrossoverPoints,
crossoverProbability)
            offspring.extend((child0, child1))
        return offspring

    def initialize(self, population):
        prototype = self._prototype
        for i in range(len(population)):
            population[i] = prototype.makeNewFromPrototype()

    def reform(self):
        random.seed(round(time() * 1000))
        np.random.seed(int(time()))
        if self._crossoverProbability < 95:
            self._crossoverProbability += 1.0
        elif self._mutationProbability < 30:
            self._mutationProbability += 1.0

    def run(self, maxRepeat=9999, minFitness=0.999):
        mutationSize = self._mutationSize
        mutationProbability = self._mutationProbability
        nonDominatedSorting = self.nonDominatedSorting
        selection = self.selection
        populationSize = self._populationSize
        population = populationSize * [None]
        self.initialize(population)
        random.seed(round(time() * 1000))
        np.random.seed(int(time()))
        currentGeneration = 0
        repeat = 0
```

```
        lastBestFit = 0.0
        while True:
            if currentGeneration > 0:
                best = self.result
                print("Fitness:", "{:f}\t".format(best.fitness), "Generation:",
currentGeneration, end="\r")
                if best.fitness > minFitness:
                    break
                difference = abs(best.fitness - lastBestFit)
                if difference <= 0.0000001:
                    repeat += 1
                else:
                    repeat = 0
                self._repeatRatio = repeat * 100 / maxRepeat
                if repeat > (maxRepeat / 100):
                    self.reform()
            offspring = self.replacement(population)
            for child in offspring:
                child.mutation(mutationSize, mutationProbability)
            totalChromosome = population + offspring
            front = nonDominatedSorting(totalChromosome)
            if len(front) == 0:
                break
            population = selection(front, totalChromosome)
            self._populationSize = populationSize = len(population)
            if currentGeneration == 0:
                self._chromosomes = population
            else:
                totalChromosome = population + self._chromosomes
                newBestFront = nonDominatedSorting(totalChromosome)
                if len(newBestFront) == 0:
                    break
                self._chromosomes = selection(newBestFront, totalChromosome)
                lastBestFit = best.fitness
            currentGeneration += 1

    def __str__(self):
        return "NSGA II"
```

Додаток Б

ВІДГУК КЕРІВНИКА
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ «ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК
на магістерську роботу

Наукового керівника Мещерякова Леоніда Івановича, д.т.н., проф. каф. ПЗКС
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

студента Трифонова Артема Дмитровича
(прізвище, ім'я, по батькові)

курсу II групи 122м-21-2
спеціальності 122 Комп'ютерні науки
освітньої програми
на тему Дослідження ефективності складання розкладу занять університету на основі застосування методів штучного інтелекту

Актуальність теми Представлена магістерська кваліфікаційна робота присвячена підвищенню ефективності процесу складання розкладу занять університету. В роботі обґрунтовано вибір способу розв'язання повних перебірних задач з метою пошуку локального мінімуму за умови бюджетування часу розв'язання за допомогою генетичних алгоритмів. З огляду на це, магістерська робота характеризується актуальністю та своєчасністю.

Мета досліджень Полягає в вивчення застосування методів штучного інтелекту для складення розкладу занять університету

Коротка характеристика розділів роботи У першому розділі було розглянуто визначення поняття та характеристику процесу складання розкладу, проаналізовані методи розв'язання подібних задач, проведено порівняльний аналіз програмних інструментів задля виявлення їх слабких та сильних сторін. У другому розділі розглянуто види генетичних алгоритмів та обґрунтовано вибір способу розв'язання повних перебірних задач з метою пошуку локального мінімуму за умови бюджетування часу розв'язання. Третій розділ містить опис та реалізацію методики застосування генетичного алгоритму до розв'язання задачі складання розкладу університету.

Практичне значення роботи полягає у впровадженні алгоритмів штучного інтелекту у процес складання розкладу занять університету для покращення організації навчального процесу.

Зауваження та недоліки В роботі відсутній більш детальний порівняльний аналіз методів штучного інтелекту, що можуть бути застосовані для вирішення

задачі складання розкладу занять університету.

Висновки та оцінка Магістром було проведено аналіз та порівняння можливих методів розв’язання поставленої задачі та обрано оптимальний варіант. Під час виконання магістерської кваліфікаційної роботи студент Трифонов А.Д. проявив себе грамотним, кваліфікованим спеціалістом, здатним приймати самостійно складні технічні рішення. Вважаю, що магістерська кваліфікаційна робота заслуговує оцінку «відмінно», а Трифонов А.Д. – присвоєння кваліфікації «магістра» з комп’ютерних наук.

Науковий керівник Мещеряков Л.І., док. техн. наук, проф., проф. каф. ПЗКС

(прізвище, ім’я, по батькові, посада, місце роботи)

«___»______________20__ р. ____________________

(підпис)

Додаток В

РЕЦЕНЗІЯ
на магістерську роботу

студента Трифонова Артема Дмитровича
(прізвище, ім'я, по батькові)

курсу II групи 122м-21-2

кафедри програмного забезпечення комп'ютерних систем

спеціальності 122 Комп'ютерні науки

освітньої програми

Тема роботи Дослідження ефективності складання розкладу занять університету на основі застосування методів штучного інтелекту

Стисла характеристика розділів роботи В першому розділі надано загальний опис наукової галузі, в якій ведеться дослідження, та наведені короткий огляд проблеми та потенційні шляхи її вирішення. Другий розділ містить опис математичних методів розв'язання задачі. Третій розділ містить опис та реалізацію методики застосування генетичного алгоритму до розв'язання задачі складання розкладу університету.

Пропозиції, внесені студентом, рівень їх наукового обґрунтування В даній кваліфікаційній роботі студентом надано декілька пропозицій щодо вирішення поставлених задач. Кожна з пропозицій була обґрунтована та підкріплена науковими даними.

Практичне значення роботи Результати роботи можуть бути застосовані для подальших наукових досліджень в даній сфері, а також вони можуть бути корисними для практичного використання.

Якість оформлення роботи Магістерська кваліфікаційна робота, яку подано на рецензію, виконана у повному обсязі у встановлений термін. Робота є добре структурованою та достатньо проілюстрованою. Викладена основна суть проблеми, що вирішується в ході виконання роботи, і шляхів її вирішення.

Недоліки в роботі відсутність зручного користувацького інтерфейсу програми та підтримки різних форматів вхідних даних. Проте вказаний недолік не впливає на позитивне враження від роботи.

Загальний висновок Магістерська кваліфікаційна робота виконана у
(підготовленість студента до самостійної роботи як спеціаліста)
відповідності з завданням із дотриманням всіх вимог.

Оцінка магістерської роботи Робота заслуговує оцінки «відмінно», а студент Трифонов А.Д. – присвоєння кваліфікації «магістра» з комп'ютерних наук.

Рецензент ____________________
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«___»______________20__ р. ____________________
(підпис)

Додаток Г

ПЕРЕЛІКЛІК ФАЙЛІВ НА ДИСКУ

Ім’я файлу	**Опис**
Пояснювальні документи	
Трифонов.doc	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Трифонов.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF.
Програма	
GA_program.rar	Архів. Містить коди програми і откомпільовану програму.
Презентація	
Презентація_Трифонов.ppt	Презентація кваліфікаційної роботи.