МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«Інформаційна технологія адаптації генетичного алгоритму для вибору стратегій в біматричних іграх»

Завідувач

випускаючої кафедри **Довбиш А. С.**

Керівник роботи **Шаповалов С. П.**

Студент групи ІНмз-01С **Петруша І. О.**

СУМИ 2021

Сумський державний університет

(назва вузу)

Факультет *ЦЗДВФН* Кафедра *Комп'ютерних наук*

Спеціальність *«122 Комп'ютерні науки»*

Затверджую:

зав.кафедрою _____

_____

"_____" _____ 20___р.

ЗАВДАННЯ

НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

Петруші Івану Олеговичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *«Інформаційна технологія адаптації генетичного алгоритму для вибору стратегій в біматричних іграх»*

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

2. Термін здачі студентом закінченого проекту (роботи) _____

3. Вхідні данні до проекту (роботи) _____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Аналіз проблеми. Постановка задачі дослідження. 2) Інформаційний огляд. 3) Математична модель та вибір методу рішення 4) Розробка інформаційного та програмного забезпечення системи*

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

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	*Аналіз проблеми. Постановка задачі дослідження*		
2.	*Інформаційний огляд*		
3.	*Математична модель та вибір методу рішення*		
4.	*Розробка інформаційного та програмного забезпечення системи*		
5.	*Оформлення пояснювальної записки до дипломної роботи*		

Студент – дипломник _____

(підпис)

Керівник проекту _____

РЕФЕРАТ

Записка: 48 сторінок., 12 рис., 2 таблиці, 2 додатки, 14 джерел інформації.

Об’єкт дослідження – неантагоністичні моделі теорії ігор.

Мета роботи – застосувати генетичний алгоритм для рішення задач теорії ігор з урахуванням різної інформаційної структури гравців.

Методи дослідження – методи математичного моделювання, адаптація генетичного алгоритму, комп’ютерна реалізація та аналіз результатів обчислення.

Результати – проведені дослідження по застосовності генетичного алгоритму для рішення задач теорії ігор. Проведена комп’ютерна реалізація та її тестування на конкретних завданнях з застосуванням мови програмування java.

НЕАНТАГОНІСТИЧНІ МОДЕЛІ ТЕОРІЇ ІГОР,
ГЕНЕТИЧНИЙ АЛГОРИТМ, КОМП’ЮТЕРНА РЕАЛІЗАЦІЯ,
ТЕСТОВІ РОЗРАХУНКИ

ЗМІСТ

ВСТУП

Завдання економічного управління по своїй суті потребує вибір оптимального рішення з множини можливих варіантів [1]. В сучасних економічних відносинах на цей вибір впливає не тільки компетентність осіб, що цим займаються, а й наявність конкурентів та фірм, що займаються в подібних сферах. Тобто конкуренція та взаємодії впливають на цей вибір. Потрібні підходи, що дозволять моделювати такі ситуації, та алгоритми і методи, що їх розв'язують.

Моделі теорії ігор — це математичне моделювання соціально-економічних явищ, що демонструє людську взаємодію, тобто конфлікт і співпрацю між особами, які приймають рішення (гравцями) [2-12].

Одночасне виникнення стратегічних, стохастичних і динамічних явищ, вплив фундаментальних некерованих аспектів, таких як знання та інформація, а також вплив інституційних та організаційних структур, роблять аналіз теоретичних ігор надзвичайно складним завданням [11].

Існує багато питань, що стосуються обчислення рівноваги в загальних біматричних іграх, що залишаються відкритими й на сьогодення, демонструючи таким чином великий розрив між труднощами в розумінні моделювання конфліктів в модель гри та подальше її розв'язання.

Центральне обчислювальне питання з точки зору інформаційних технологій – чи існує алгоритм, час роботи якого вимірюється в термінах довжини введення, такий, що виконується за поліноміальний час у вхідному розмірі чи ні.

У 2004 році Стенгель і Савані [11] провели дослідження на відомому алгоритму Лемке-Ховсона й довели, що для знаходження рівноваги Неша цей алгоритм не є алгоритмом поліноміального часу.

Перспективним напрямком в розв'язанні конфліктів є аналіз застосовності відомих чисельних методів та алгоритмів [3-8].

Пошук нових підходів та адаптація відомих алгоритмів для рішення моделей теорії ігор залишаються актуальними. Особливо цікавим є питання застосовності еволюційних алгоритмів для рішення поставленої проблеми.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

Спеціалісти та наукові фірми в області прийняття рішень в моделях конфліктів виконують методологічно ті ж дослідження, що відносять до традиційних при прийнятті рішень [1]. На основі вхідної інформації будується модель теорії ігор, до рішення якої залучається відомі алгоритми та створюється інформаційно-програмне забезпечення, метою якого є надання допомоги особам, що приймають рішення в ситуаціях, що досліджуються [2-12]. У деяких аспектах теорія ігор є наукою про стратегію або, принаймні, про оптимальне прийняття рішень [1].

Використання теоретико-ігрової конструкції для опису тієї чи іншої групи соціально-економічних явищ слід віднести до концептуального аспекту математичного забезпечення, яке полягає у формуванні вихідної системи базових понять та встановленні в ньому такої ієрархічної впорядкованості та логічної структури, яка б піддавалася уявленню в математичному вигляді.

Рисунок 1.1 – Одна із можливих класифікацій моделей теорії ігор

Теоретико-ігрову конструкцію представимо категорійною моделлю ⟨ N, A, K, P ⟩, що складається з:

- N – кількість гравців (учасників конфліктної ситуації) гри;
- A – множина альтернативних виборів (стратегій) які можуть гравці застосовувати у грі;

- К – кількісна оцінка (функція виграшу, корисність для кожного гравця) за допомогою якої характеризується ситуація гри, що виникає при виборі гравцями стратегій з множини можливих ($Ki\colon \prod A_i \rightarrow R$)
- Р – переваги вибору таких стратегій, що приведуть до одержання найкращого або компромісного (на яке будуть згодні обидві сторони конфлікту) результату гри.

Використовуючи інструментарій теорії ігор, можна викласти реальні сценарії для таких ситуацій, як цінова конкуренція та випуск продуктів (і багато інших), і передбачити їхні результати.

1.1 Неантагоністичні моделі теорії ігор

У теорії ігор біматрична гра — це одночасна гра для двох гравців, в якій кожен гравець має скінченну кількість можливих дій. Назва походить від того факту, що звичайну форму такої гри можна описати двома матрицями — матрицею A, з елементами a_{ij}, що описує виграші гравця 1, і матрицею B, з елементами b_{ij}, що описує виграші гравця 2. Нехай маємо неантагоністичну конфліктну ситуацію двох гравців A і B, кожен з яких має в цій грі матрицю виграшів:

$$\begin{pmatrix} a_{11}a_{12}...a_{1n} \\ a_{21}a_{22}...a_{2n} \\ .... \\ a_{m1}a_{m2}...a_{mn} \end{pmatrix} \quad \ldots\ldots \quad (1.\ 1)$$

$$\begin{pmatrix} b_{11}b_{12}...b_{1n} \\ b_{21}b_{22}...b_{2n} \\ .... \\ b_{m1}b_{m2}...b_{mn} \end{pmatrix} \quad (1.\ 2),$$

Відмінність біматричних ігор полягає також в тому, що окрім рівноваги Неша (РН) та рівноваги в домінуючих стратегіях, в цих іграх можливі ще рівновага Парето (РП) та рівновага Штакельберга (РШ). Слід додати, що і рівноваг

Неша в біматричній грі може бути декілька, що приводить до додаткових труднощів в виборі оптимальних рішень.

Зацважим що, хоча біматрична гра відноситься до класу безкоаліційних ігор, в деяких моделях виникає спокуса укладання між гравцями угод щодо вибору ситуацій в грі з додатковим розподілом між собою виграшів (щось на зразок коаліційних угод), або узгодження між собою дій без додаткових платежів (при декількох РН в грі). Якщо ж перемовини між гравцями неможливі або заборонені, то кожний гравець грає на свій розсуд та спираючись на інформацію, що йому відома. Іноді в такій ситуації діє принцип «невидимої руки Адама Сміта», що заспокоює суспільство в тому, що це може бути для нього і краще, ніж гравці мають право на домовленість.

Принцип «невидимої руки Адама Сміта». *«Вільний ринок», незважаючи на те, що кожен з «ринкових агентів» максимізує власний прибуток, сприяє росту в цілому суспільного багатства, підвищує ефективність виробництва.*

Основне призначення цього принципу – економіка, але його можна віднести з деякою долею скептизму і до теорії ігор.

Перейдемо до алгоритмів пошуку різного виду рівноваги.

Основна ідея пошуку залишається незмінною – потрібно знайти таку ситуацію гри, яку жодному гравцеві невигідно залишати.

Рівновага Неша (РН).

Алгоритм пошуку РН в чистих стратегіях.

- Для першого гравця обираємо його найкращі відповіді на дії другого гравця. З цією метою в кожному стовпчику (фіксуємо дію другого гравця) матриці (1.1) відмічаємо (підкреслюємо) найбільший виграш серед a_{ij}.
- Для другого гравця проводимо аналогічні дії, але тепер фіксуємо кожен рядок та відмічаємо в ній найбільший виграш b_{ij} для другого гравця.
- Якщо у вихідній матриці одержимо ситуацію, коли в будь якого її подвійного елемента (a_{ij},b_{ij}) будуть відмічені обидва виграші, то значить у грі відбудеться рівновага Неша в чистих стратегіях, які представляють собою

рядок для першого гравця та стовпчик – для другого, на перетині яких знаходиться відмічений подвійний елемент.

- Якщо вказаної вище ситуації не відбулося, то РН в чистих стратегіях для даної гри не існує.

Зауважимо, що біматрична гра може мати декілька рівноваг Неша в чистих стратегіях або зовсім може не мати таких.

Коли потрібно шукати рівновагу Неша в змішаних стратегіях, то це і є предметом досліджень в роботі.

Приклад пошуку РН в грі. Нехай задано матрицю гри (дивись табл. 1.1)

Таблиця 1.1 Представлення гри матрицею

	К	Л	М
А	2,3	-4,-1	-5,4
Б	-1,-3	0,-2	1,-4
В	3,2	-2,-1	-3,1

В цій грі, за описаним вище алгоритмом, маємо 2 сукупності стратегій, які задають рівновагу Неша – це подвійно-відмічені стратегії (Б, Л) та (В,К). Очевидно, що ситуація (В,К) вигідніша ніж (Б, Л) для обох гравців.

Та навіть в такій очевидній на перший погляд грі в плані вибору РН все ж попередня домовленість не буде зайвою.

Рівновага Парето (РП).

Рівновага Парето (РП) –це така ситуація в грі, з якої не можливо вийти жодному гравцеві з метою збільшити свій виграш, не зменшуючи при цьому виграші інших учасників гри

Алгоритм пошуку РП в чистих стратегіях.

- Вибираємо певний подвійний елемент матриці (1.1), що задає виграші гравців, і, перебираючи всі інші елементи, порівнюємо значення виграшів гравців між собою. Якщо серед них знайдеться такий, коли обидва гравців поліпшують свої виграші в порівнянні з вибраним, то переходимо до іншого елементу, бо за означенням РП не відбулося. І таким чином перебираємо всі елементи матриці (1.1).

- Якщо в матриці (1.1) знайдеться такий подвійний елемент, що у всіх інших поліпшення (збільшення!) виграшу одного із гравців можливе лише тільки за умови погіршення (зменшення!) виграшу іншого гравця, то ми знайшли стратегії, які задають оптимум Парето.

Щоб зрозуміти сутність оптимуму Парето, розглянемо множину точок Ω на площині (U,V). Точки цієї множини розподіляються на внутрішні (такі точки які належать, разом з точками будь якої малої округи) та граничні точки (в будь якій околиці яких є і внутрішні точки і точки , що не належать Ω). Точки множини Ω розіб'ємо на 3 класи (див. рис.1. 2) :

1. До першого класу відносяться точки, які можна зрушити з місця так, щоб збільшити при цьому обидві її координати і при цьому точка залишиться в множині Ω (до цього класу відносяться всі внутрішні точки і частина граничних);
2. Другий клас утворюють точки, зміщенням яких по множині Ω можна збільшити тільки одну з координат при зберіганні значення другої (вертикальний відрізок AB та горизонтальний відрізок PQ на границі множини).
3. В третій клас потрапляють такі точки (точки дуги BQ границі області), зміщення яких по множині Ω не в змозі збільшити жодної координати.

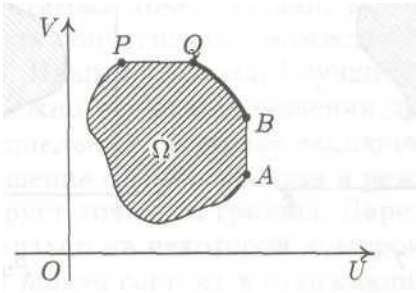

Рисунок 1.2 – Множина точок області Ω

Множину точок третього класу називають множиною Парето даної множини Ω.

Рівновага Штакельберга.

Рівновагою Штакельберга (РШ) з лідером, який має номер 1, називається така сукупність стратегій всіх інших гравців, що перший гравець (лідер) з урахуванням цілей інших гравців прогнозує рівновагу Неша, яка буде складатися між

ними після його ходу, і відповідно до цього оптимізує свою стратегію. Інші ж гравці вибирають стратегію відповідно до його прогнозу.

Загалом же і рівновага Штакельберга, і рішення Штакельберга має припущенням, що ми вірно можемо змоделювати поведінку інших (тобто перебіг гри, який буде мати місце після рішення лідера). Особливо наглядно рішення Штакельберга та рівновага Штакельберга визначаються для у випадку лише двох гравців. У цьому випадку алгоритм знаходження рішення Штакельберга для першого гравця може бути записано таким чином. 1. Розглядаємо першу стратегію лідера. Вибираємо той виграш, який є найбільшим для другого гравця. Відмічаємо відповідний виграш лідера (тобто той, який знаходиться в тій же комірчині).
2. Проводимо аналогічні дії для кожної стратегії лідера.
3. Розглядаємо множину всіх виграшів лідера, які ми відмітили, і вибираємо найбільший. Стратегія, що відповідає цьому елементу і є рішенням Штакельберга.

1.2 Постановка задачі

Прийдемо до наступної постановки задачі.

1. Сформулювати математичну модель біматричної гри двох гравців.
2. Вибрати генетичний алгоритм в якості інструментарію для рішення створеної моделі.
3. Адаптувати генетичний алгоритм для вирішення поставленої задачі.
4. Створити інформаційне й програмне забезпечення по розв’язанню біматричних ігор за допомогою генетичного алгоритму.
5. Провести тестові дослідження та проаналізувати одержані результати на предмет подальшого впровадження в процес прийняття рішень.

2 МОДЕЛЬ ТЕОРІЇ ІГОР ТА АЛГОРИТМ РІШЕННЯ

2.1 Загальна модель дослідження

Розглянемо скінчену безкоаліційну гру для двох осіб. Зазвичай в такій грі задають дві матриці однакового розміру виграшів першого і другого гравців. Строки цих матриць відповідають стратегіям першого гравця, а стовпці матриць - стратегіям другого гравця . При цьому в першій матриці представлені виграші першого гравця, а в другій матриці - виграші другого.

Гра, що задається сукупністю об'єктів $\{X, Y, K_x, K_y\}$, де X, Y – непусті множини і функції $K_x : X \times Y \to R$, $K_y : X \times Y \to R$, називається біматричною грою. Елементи $x \in X$, $y \in Y$ називаються стратегіями гравців 1 і 2, елементи декартового добутку $X \times Y$ - ситуаціями, функції K_X, K_Y – функціями виграшів гравців 1 та 2 відповідно.

В загальному разі біматрична гра – це гра з ненульовою сумою і кожний гравець одержує в ній свій приз (максимізує виграш). Вважаємо, що виграш є дійсним числом, яке показує ступінь досягнення бажаного результату.

Так як матриці виграшів гравців одного порядку (m×n), то біматричну гру представимо однією матрицею (таблицею), кожний елемент якої складається з двох чисел, розділених комою, перше з яких показує виграш першого гравця, друге – другого

$$G = \begin{pmatrix} a_{11}, b_{11} & a_{12}, b_{12} & \dots & a_{1n}, b_{1n} \\ a_{21}, b_{21} & a_{22}, b_{22} & \dots & a_{2n}, b_{2n} \\ \dots\dots & \dots\dots & \dots\dots & \dots\dots \\ a_{m1}, b_{m1} & a_{m2}, b_{m2} & \dots & a_{mn}, b_{mn} \end{pmatrix} \qquad (2.1)$$

При дослідженнях біматричних ігор постають ті ж самі питання теорії ігор: які оптимальні стратегії гравці повинні обрати в грі і які виграші при цьому вони одержать?

Відповідь на це питання – внаслідок того, що кожен з гравців прагне одержати максимальну корисність (виграш) в грі, потрібно побудувати таке (компромісне) рішення, яке в тому чи іншому змісті задовольнить обох гравців.

Перейдемо до алгоритмів пошуку різного виду рівноваги.

Основна ідея пошуку залишається незмінною – потрібно знайти таку ситуацію гри, яку жодному гравцеві невигідно залишати.

Принципи пошуку рівноваги Неша (РН) залишаються незмінними, визначення РН має такий самий зміст:

Для гри $G_N = \{S_i, K_i, i \in N\}$, стратегії $(s^*_1, s^*_2, \ldots s^*_N)$ утворюють рівновагу Неша, якщо для кожного гравця і стратегія s^*_i є його найкращою відповіддю-реакцією на набір стратегій s^*_{-i} інших гравців, тобто s^*_i розв'язує проблему

$$\max_{s_i \in S_i} K_i(s_1^*, \ldots, s_{i-1}^*, s_i, s_{i+1}^*, \ldots, s_n^*)$$

Зрозуміло, що якщо РН в чистих стратегіях в грі не відбувається та домовленості між гравцями не існує, гра стає нестійкою і гравцям слід розширити пошук рішення на змішані стратегії:

$S_A(p_1, p_2, \ldots p_m)$, $S_B(q_1, q_2, \ldots q_n)$. Тоді середні виграші гравців А і В знаходимо відповідно (математичні сподівання):

$$E_A(A, S_A, S_B) = \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij} p_i q_j, \quad E_B(B, S_A, S_B) = \sum_{i=1}^{m}\sum_{j=1}^{n} b_{ij} p_i q_j,$$
$$\sum_{i=1}^{m} p_i = 1, \quad \sum_{j=1}^{n} q_j = 1 \qquad (2.2)$$

Стратегії $S^*_A(p^*_1, p^*_2, \ldots p^*_m)$, $S^*_B(q^*_1, q^*_2, \ldots q^*_n)$ визначають рівноважну ситуацію, якщо одночасно виконуються нерівності

$$E_A(A, S^*_A, S^*_B) \geq E_A(A, S_A, S^*_B); \; E_B(B, S^*_A, S^*_B) \geq E_B(B, S^*_A, S_B) \qquad (2.3)$$

Маємо математичну модель для пошуку РН в змішаних стратегіях гри двох осіб з довільною сумою, що складається з матриці гри (2.1), співвідношень (2.2) та нерівностей (2.3). Проблема пошуку РН тісно пов'язана з аналізом найкращих відповідей $R_i(s_{-i})$ гравця і на профіль стратегій s_{-i} на профіль стратегій інших гравців.

Складемо профіль найкращих відповідей всіх гравців $R(s) = (R^i(s_{-i}), i \in N)$. Зрозуміло, що s^* є РН тоді і тільки тоді, коли $s^* \in R(s^*)$. Таким чином існування

РН тісно пов’язане з існуванням нерухомих точок відображення найкращих відповідей .

Оскільки середні виграші гравців задані функціями, що представляють полілінійні форми (2. 2), то якщо зафіксувати змішані стратегії одного з гравців, середній виграш іншого стає лінійною функцією відносно його змішаних стратегій, а отже максимум цієї функції досягається в крайніх точках, тобто на чистих стратегіях. Тому, потрібно взяти всі найкращі відповіді в чистих стратегіях, та взяти всі змішані стратегії, котрі приписують ненульові ймовірності тільки оптимальним відповідям в чистих стратегіях.

Алгоритм пошуку РН в змішаних стратегіях.

- Для кожного гравця виділяється деяка підмножина $S_{0i} \subseteq S_i$ чистих стратегій і складається система рівнянь

  $E_i(s_i,\mu\text{-}i) = c_i$, $\forall\ s_i \in S_{0i}$, $i \in N$. В цій системі змінними є числа c_i та ймовірності $\mu_j(s_j)$ при $s_j \in S_{0j}$. Інші $\mu_j(s_j)$ приймаються рівними 0, враховуючи умову

  $$\sum_{s_i \in S_i} \mu_i(s_i) = 1$$

  Якщо гравців тільки двоє, то система є лінійною, що обговорювалось раніше. Якщо гравців більше, то рішення такої нелінійної системи ускладнюється.

- Після знаходження рішення потрібно перевірити ймовірності на невід’ємність та умову найкращої відповіді

  $E_i(s_i,\mu\text{-}_i) \le c_i$. Якщо всі умови виконані, то РН в змішаних стратегіях знайдено, якщо ні, то переходимо до іншої підмножини S_{0i} .

У більш загальному випадку біматрічной гри ситуація рівноваги по Нешу обчислюється з використанням різних лінійних методів, що мають своєю основою завдання лінійного програмування. Історично, одна з перших підходів розроблений на початку 60-х років. Це відомий алгоритм Лемке - Хаусона знаходження рішення в біматрічной грі [10-11]. У ситуації рівноваги гри двох осіб мішана стратегія одного гравця урівноважує виграш другого гравця при використанні їм чистих стратегій.

2.2 Вибір методу рішення

Вчені починають все більше розуміти, що природа є чудовим джерелом натхнення для розробки інтелектуальних систем і алгоритмів. У сфері обчислювального інтелекту, особливо еволюційних обчислень і систем, заснованих на імітації процесу рішення як це відбувається в природі такі дослідження завойовують все більше простору. Ми знаходимося на порозі розвитку та пропонування нових алгоритмів та/або систем, які частково або повністю відповідають природі та її діям та реакціям, що відбуваються в конкретній природній системі чи виді.

В дослідженнях інформаційних технологій та Computer Science генетичний алгоритм (GA) — це метаевристика, натхненна процесом природного відбору, що належить до великого класу еволюційних алгоритмів (EA) [8]. Генетичні алгоритми зазвичай використовуються для створення високоякісних рішень проблем оптимізації та пошуку, покладаючись на біологічно направлені оператори, такі як мутація, кросовер і відбір. Деякі дослідження, що базуються на використанні GA включають оптимізацію дерев рішень для кращої продуктивності, автоматичне розв’язування головоломок судоку, оптимізацію гіперпараметрів тощо.

У генетичному алгоритмі популяція варіантів рішень (так звані особини, істоти або фенотипи) оптимізаційної проблеми розвивається від початкових до кращих рішень. Кожне рішення-кандидат має набір властивостей (його хромосоми або генотип), які можуть бути мутовані та змінені; традиційно рішення представлені в двійковому вигляді у вигляді рядків з 0 і 1, але можливі й інші кодування [8].

Еволюція зазвичай починається з популяції випадково згенерованих індивідів і є ітераційним процесом, при цьому популяція на кожній ітерації називається поколінням. У кожному поколінні оцінюється придатність кожної особини в популяції; придатність – це зазвичай значення цільової функції в оптимізаційній задачі, що вирішується. Більш придатні особини стохастично відбираються з поточної популяції, і геном кожної людини модифікується (рекомбінується і, можливо, випадково мутується) для формування нового покоління. Нове покоління

варіантів рішень потім використовується на наступній ітерації алгоритму. Зазвичай алгоритм припиняє роботу, коли або створено максимальну кількість поколінь, або досягнуто задовільного рівня придатності для популяції, іноді умовою зупинення послуговує час, відведений на еволюційний процес в цілому.

Типовий генетичний алгоритм вимагає: генетична репрезентація домену рішення, функція пристосованості для оцінки області рішення.

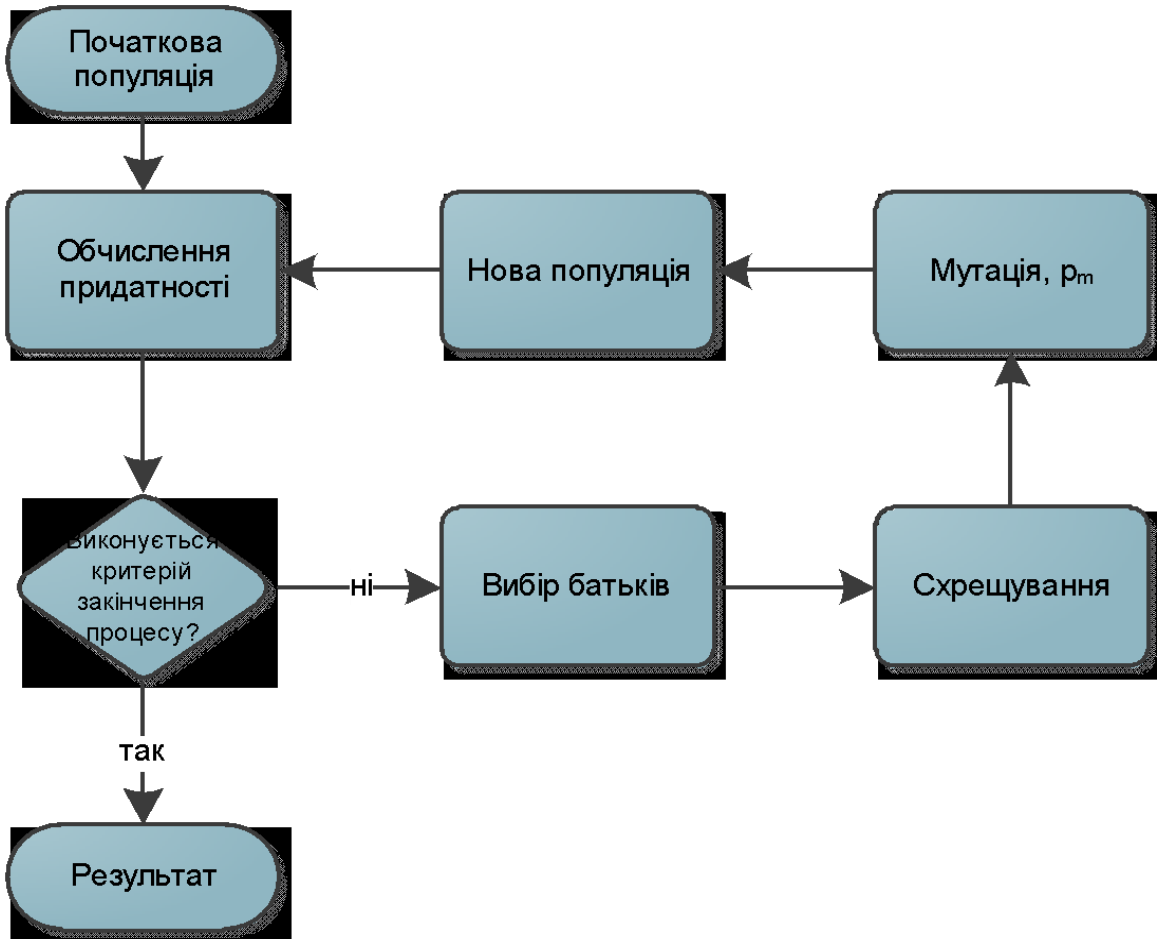

Рисунок 2.1 – Класична схема генетичного алгоритму

Стандартне представлення кожного рішення-кандидата у вигляді масиву бітів (також званого бітовим набором або бітовим рядком).[3] Масиви інших типів і структур можна використовувати, по суті, таким же чином. Основна властивість, яка робить ці генетичні уявлення зручними, полягає в тому, що їх частини легко вирівнюються завдяки фіксованому розміру, що полегшує прості операції кросинговеру. Також можуть використовуватися представлення змінної довжини, але в цьому випадку реалізація кросовера є більш складною. Деревоподібні уявлення досліджуються в генетичному програмуванні, а уявлення у формі

графів досліджуються в еволюційному програмуванні; поєднання лінійних хромосом і дерев досліджується в програмуванні експресії генів.

Після визначення генетичного представлення та функції придатності GA переходить до ініціалізації популяції рішень, а потім до її покращення за допомогою багаторазового застосування операторів мутації, кросинговеру, інверсії та відбору.

Розглянемо основні блоки алгоритму та їх призначення.

1. *Ініціалізація.*

Розмір популяції залежить від характеру проблеми, але зазвичай містить кілька сотень або тисяч можливих рішень. Часто початкова сукупність генерується випадковим чином, що дозволяє отримати весь діапазон можливих рішень (простір пошуку). Іноді рішення можуть бути «засіяні» в областях, де, ймовірно, будуть знайдені оптимальні рішення.

2. *Відбір.*

Протягом кожного наступного покоління відбирається частина існуючої популяції для виведення нового покоління. Індивідуальні рішення вибираються за допомогою процесу, що ґрунтується на придатності, де, як правило, з більшою ймовірністю будуть обрані кращі рішення (виміряні функцією придатності). Певні методи відбору оцінюють придатність кожного рішення і переважно вибирають найкращі рішення. Інші методи оцінюють лише випадкову вибірку сукупності, оскільки перший процес може зайняти дуже багато часу.

3. *Fitness – функція (функція пристосованості).*

Дійсна або цілочисельна функція однієї або декількох змінних, яка спрямовує еволюцію у бік оптимального рішення.

Функція пристосованості визначається за генетичним уявленням і вимірює якість представленого рішення. Функція фітнесу завжди залежить від проблеми.

4. *Генетичні оператори.*

Основні оператори: кросовер (генетична рекомбінація) і мутація (генетичне відновлення).

Хоча кросовер і мутація відомі як основні генетичні оператори, у генетичних алгоритмах можна використовувати інші оператори, такі як перегрупування, колонізація-вимирання або міграція.

Генетичні оператори послуговують створенню популяції другого покоління рішень із тих, які відібрано за допомогою комбінації генетичних операторів: кросовер (також званий рекомбінацією) та мутація.

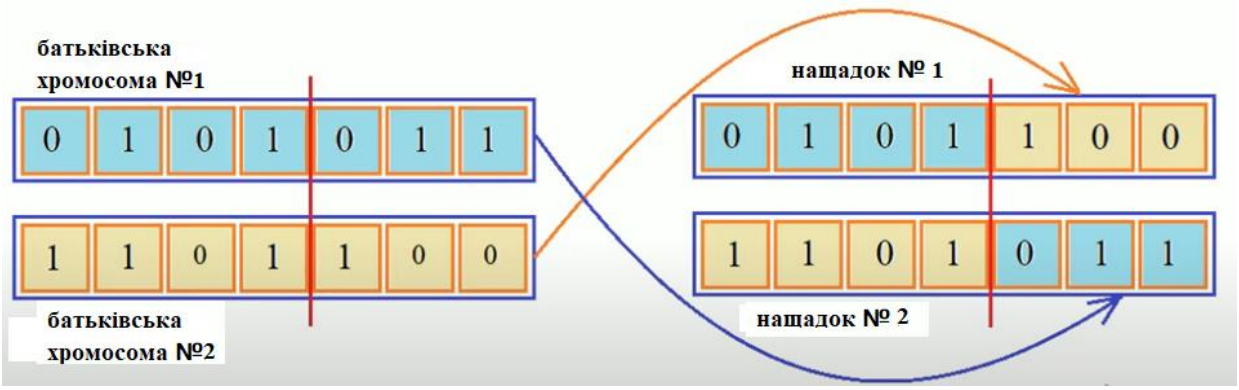

Рисунок 2.2 – Операція схрещування

Створюючи «дочірнє» рішення з використанням вищевказаних методів кросинговеру та мутації, створюється нове рішення, яке зазвичай поділяє багато характеристик своїх «батьків». Для кожної нової дитини обираються нові батьки, і процес триває до тих пір, поки не буде створено нову сукупність рішень відповідного розміру. Хоча методи розмноження, які ґрунтуються на використанні двох батьків, більше «надихаються біологією», деякі дослідження показують, що більше двох «батьків» генерують хромосоми більш високої якості.

Як правило, середня пристосованість підвищиться завдяки цій процедурі для популяції, оскільки для розведення відбираються лише найкращі організми з першого покоління, а також невелика частка менш придатних розчинів. Ці менш відповідні рішення забезпечують генетичне різноманіття в генетичному фонді батьків і, отже, забезпечують генетичну різноманітність наступного покоління дітей.

Думки щодо важливості схрещування проти мутації розділилися. У Fogel (2006) є багато посилань, які підтверджують важливість пошуку на основі мутацій.

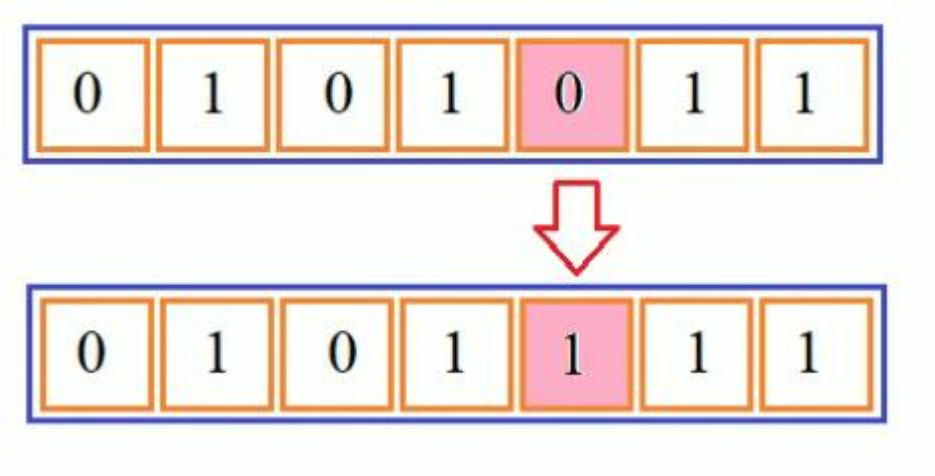

Рисунок 2.3 – Операція мутація

Саме мутація розширити область пошуку рішень задачі та зберігає різноманітність популяції

Варто налаштувати такі параметри, як ймовірність мутації, ймовірність кросинговеру та розмір популяції, щоб знайти розумні налаштування для класу

5. *Умова останову алгоритму*

Процес генерації повторюється, поки не буде досягнута умова припинення роботи алгоритму. Поширені умови припинення:

- Знайдено рішення, яке задовольняє мінімальним критеріям.
- Досягнута фіксована кількість поколінь.
- Призначений бюджет (час на обчислення/гроші) досягнуто.
- Придатність рішення з найвищим рейтингом досягла або досягла такого плато, що послідовні ітерації більше не дають кращих результатів.
- Ручна перевірка.
- Комбінації перерахованого вище.

Генетичні алгоритми є порівняльно новим напрямком в інформаційних технологіях та Computer Science. Вони здатні не тільки вирішувати і скорочувати перебір у складних завданнях, але й легко адаптуватися до зміни проблеми, як наприклад до проблеми теорії ігор.

2.3 Адаптація генетичного алгоритму для рішення задач теорії ігор

Гра насамперед представляє собою процес, в якому сукупність визначених методів взаємодії гравців приводить до результату. Кожна зі сторін використовує дії (стратегії) з множини можливих, щоб вести гру до максимізації корисності й своїми діями впливає на корисність супротивника. Багаторазове виконання гри рано чи пізно приведе гравців в точку рівноваги.

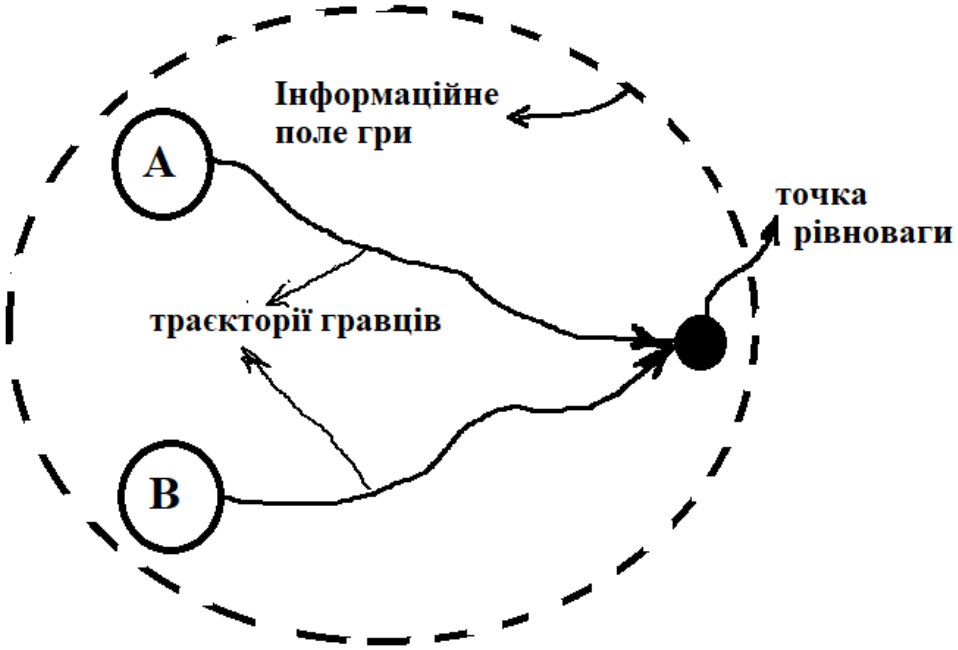

Рисунок 2.4 – Сходження траєкторій до точки рівноваги

«Ігрова діяльність» дозволяє отримати та розвинути навички у вирішенні конфліктних ситуацій й розуміти гру, як еволюційний процес. Тобто гра є одним із способів пізнавальної діяльності людини в оточуючому просторі. Й точку рівноваги в грі можна розуміти як одержання такої ситуації, в яку гравці прийшли в процесі еволюції своїх навиків та розумінь.

З вищенаведених позицій, застосування еволюційних алгоритмів, до яких відноситься генетичний алгоритм, в дослідженнях конфліктів є природним й самим собою зрозумілим.

Будемо розглядати гру, як еволюційний процес, в ході якого гравці пристосовуються до поведінки один одного й приходять рано чи пізно до рівноваги к грі.

Дослідимо застосування генетичного алгоритму в аналізі моделей теорії ігор. Будемо вбачати що і грі беруть участь дві сторони (два гравця) з наборами стратегій, що можуть у грі застосовувати. Блок схема адаптації генетичного алгоритму наведена на рисунку 2.3.

Представимо основні його блоки.

1. Початкова популяція особин (хромосом) генерується випадковим чином й складається з генів, що задають різні варіанти вибору гравцями своїх змішаних стратегій. У гравця А – це набір (множина) S_A, у гравця В – S_B:

   $S_A(p_1, p_2, p_3,\ldots,p_n)$; $S_B(q_1,q_2,q_3,\ldots,q_n)$.

   Кожна особина буде складатися з двох хромосом X та Y, в перший набір генів – набір стратегій гравця А, в другій – В.

2. Fitness функція (функція пристосованості) являє собою виграші гравців (результат гри):

$$E_A(A,S_A,S_B)=\sum_{i=1}^{m}\sum_{j=1}^{n}a_{ij}p_iq_j,\quad E_B(B,S_A,S_B)=\sum_{i=1}^{m}\sum_{j=1}^{n}b_{ij}p_iq_j,$$

$$\sum_{i=1}^{m}p_i=1,\quad \sum_{j=1}^{n}q_j=1$$

   За цією функцією відбираємо 60% кращих для схрещування, 40% - для мутації.

3. Схрещування (crossover) . Це бінарна операція, що потребує дві батьківські особини, а результатом є два нащадки, що вспадкоємлюють гени батьків.

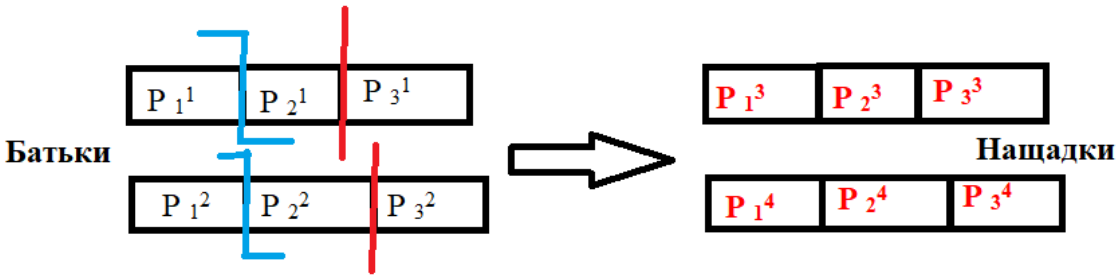

Рисунок 2.5 – Схема схрещування

Crossover відбувається за такою процедурою. В батьківських вузлах, третій ген відокремлюється й у схрещенні не приймає участь. Він потім

сам довизначається за іншими двома. Перший та другий ген батьківських вузлів змінюються місцями.

4. Мутація. Це унарна операція й відбувається за простою схемою.

Рисунок 2.6 – Схема мутації

У батьківській хромосомі обирається ген, що має найбільше значення й переноситься до нащадку на те ж саме місце. До інших генів проводиться процедура мутація шляхом додавання до одного з них (вибирається випадково) визначеного числа (0,05), а від іншого це число віднімається і вони оновлені переходять до нащадкової особини.

5. В результаті виконання генетичних операцій одержимо нове покоління, пристосовність якого перевіряється знову Fitness функцією.
6. Переходимо до блоку «Умови зупину».

   Застосовність цього блоку полягає в виборі умови, за якою дія генетичного алгоритму призупиняється й ми отримуємо результати рішення. Щонайчастіше цією умовою є кількість можливих змін поколінь, або час виконання рішення.

   Наприклад, задаємо, що в процесі еволюції потрібно щоб відбулося 100 змін покоління. Або задаємо час виконання алгоритму – 5 хвилин.
7. Якщо умова зупину виконається, то ми отримуємо рішення задачі та перевіряємо його на валідність.
8. Якщо умова зупину не відбулася, повертаємось до процедур відбору, схрещування та мутації.

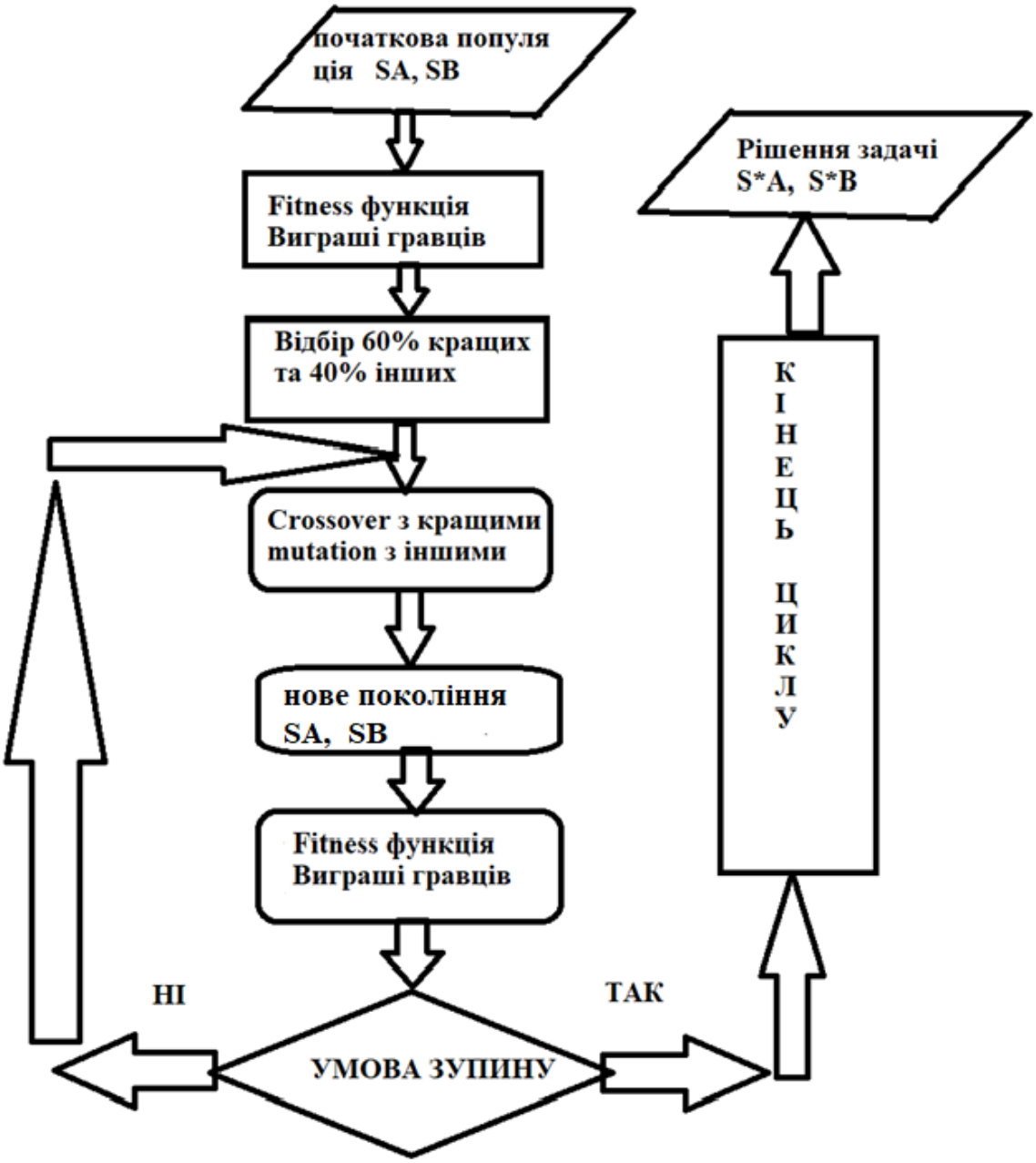

Рисунок 2.7 – Схема адаптивного генетичного алгоритму для рішення задач теорії ігор

3 КОМПЮТЕРНА РЕАЛІЗАЦІЯ ТА ТЕСТОВІ РОЗРАХУНКИ

3.1 Опис комп'ютерної реалізації

Комп'ютерна реалізація проекту виконана на алгоритмічній мові Java - об'єктно орієнтованій мові програмування високого рівня, що компілюється в байт-код і запускається на JVM [13,14].

За графіком індексу популярності мов програмування TIOBE з 2002 по 2018 рік Java стабільно була на вершині рейтингу з середини 2015 до початку 2020 року (дивись рис. 3.1).

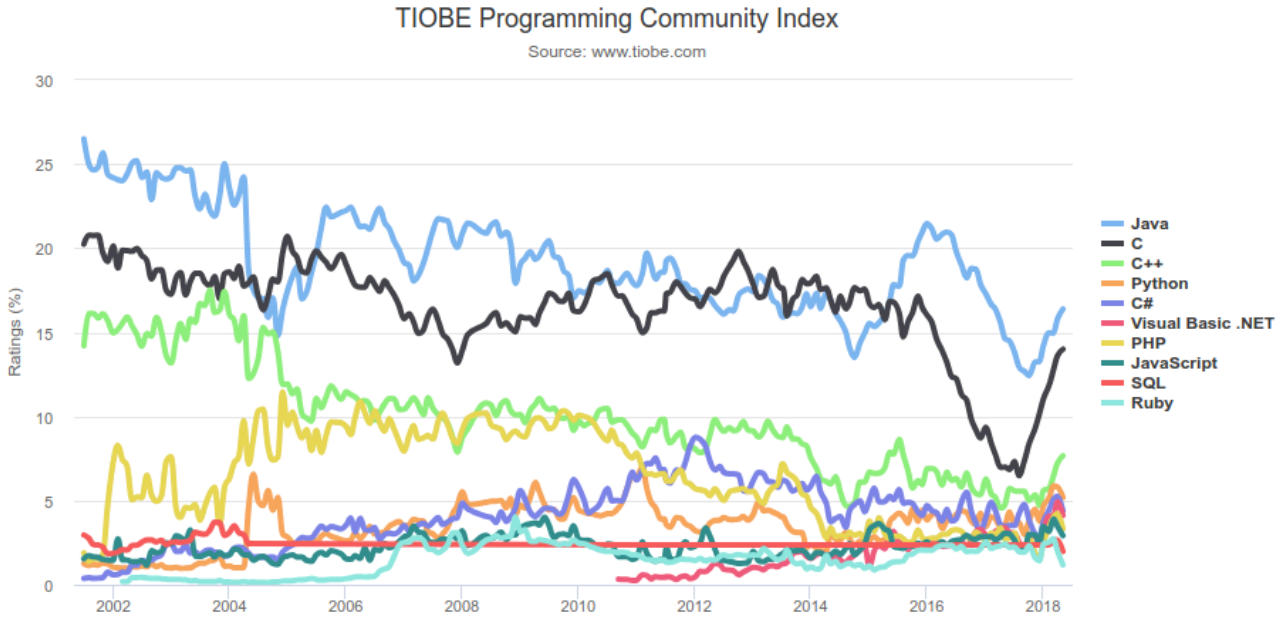

Рисунок 3.1 – Графік індексу популярності мов програмування

Java — це високорівнева, заснована на класах, об'єктно-орієнтована мова програмування, яка розроблена так, щоб мати якомога менше труднощів реалізації. Це мова програмування загального призначення, призначена для того, щоб користувачі-програмісти могли писати один раз, запускати в будь-якому місці (WORA), що означає, що скомпільований код Java може працювати на всіх платформах, які підтримують Java, без необхідності перекомпіляції.

За синтаксисом код, написаний на Java подібний до C і C++, але має менше засобів низького рівня, ніж будь-який з них. Середовище виконання Java надає динамічні можливості (такі як відображення та модифікація коду під час виконання), які зазвичай недоступні в традиційних скомпільованих мовах. Станом на

2019 рік Java була однією з найпопулярніших мов програмування, що використовуються за даними GitHub, особливо для веб-додатків клієнт-сервер, із заявленими 9 мільйонами розробників.

Станом на жовтень 2021 року Java 17 є останньою версією, Java 8, 11 і 17 є поточними версіями довгострокової підтримки (LTS). Oracle випустила останнє загальнодоступне оновлення для застарілої версії Java 8 LTS у січні 2019 року для комерційного використання, хоча в іншому випадку вона все ще підтримуватиме Java 8 з загальнодоступними оновленнями для особистого використання на невизначений термін. Інші виробники почали пропонувати збірки OpenJDK 8 і 11 за нульовою вартістю, які все ще отримують безпеку та інші оновлення.

Комп’ютерна реалізація поставленого завдання виконана за блок-схемою, приведеною в розділі 2.3 з використанням об'єктно-орієнтованого підходу, що базується на використанні класів. В об'єктно-орієнтованому програмуванні клас є розширюваним програмним кодом-шаблоном для створення об'єктів, що надає початкові значення для стану (змінні-члени) та реалізацій поведінки (функції або методів).

На практиці такий підхід привів до створення деякої кількості класів, включаючи інтерфейс та реалізацію, а також подальше їх використання.

Наприклад, створюється клас хромосом, які потім приймають участь в ініціалізації, генетичних операціях:

```
package com.company;
import java.util.Arrays;
public class Chromosome {
    private double[] genesP;
    private double[] genesK;
    private double fitness;
    public Chromosome(double[] genesP, double[] genesK, double fitness) {
        this.genesP = genesP;
        this.genesK = genesK;
        this.fitness = fitness;
```

```
    }
    public Chromosome(double[] genesP, double fitness) {
        this.genesP = genesP;
        this.fitness = fitness;
    }
    public double[] getGenesP() {
        return genesP;
    }
    public void setGenesP(double[] genesP) {
        this.genesP = genesP;
    }
    public double[] getGenesK() {
        return genesK;
    }
    public void setGenesK(double[] genesK) {
        this.genesK = genesK;
    }
    public double getFitness() {
        return fitness;
    }
    public void setFitness(double fitness) {
        this.fitness = fitness;
    }
}
```

Й сам генетичний алгоритм має свій клас, в якому наприклад ми вводимо вхідні дані

```
public class GeneticAlgorithmBimatrixGame {
    /* Матриця гравця A */
    public static double[][] valuesAMatrix = new double[][]{
            {8, 4, 2},
```

```
            {2, 8, 4},
            {1, 2, 8}
        };

        /* Матриця гравця B */
        public static double[][] valuesBMatrix = new double[][]{
            {7, 5, 1},
            {3, 7, 5},
            {0, 3, 7}
    };
```

Генерація початкової популяції особин (хромосом) здійснюється за процедурою

```
        /* Генерує набір генів хромосоми */
        private static double[] generateGenes() {
            double[] randomNumbers = ThreadLocalRandom.current().doubles(3).map(i -> round(i, 3)).toArray();
            double sumRandomNumbers = Arrays.stream(randomNumbers).sum();
            /* Кожне згенероване число ділимо на їх суму щоб отримати числа, які в сумі дадуть 1 */
            return Arrays.stream(randomNumbers).map(i -> round(i / sumRandomNumbers, 3)).toArray();
    }
```

Розрахунок Fitness-функції:

```
        /* Розраховує значення фітнес функції для хромосоми */
        private static double calculateFitness(Chromosome chromosome) {
            return calculateFitness(chromosome.getGenesP(), chromosome.getGenesK());
        }
        /* Розраховує значення фітнес функції для наборів генів */
```

```
        public static double calculateFitness(double[] genesP, double[] genesK) {
            double fitness = 0;
            for (int i = 0; i < genesP.length; i++) {
               for (int j = 0; j < genesK.length; j++) {
                  fitness += GeneticAlgorithmBimatrixGame.valuesAMatrix[i][j] * genesP[i] * genesK[j];
                  fitness += GeneticAlgorithmBimatrixGame.valuesBMatrix[i][j] * genesP[i] * genesK[j];
               }
        }
```

3.2 Тестові розрахунки

Для перевірки дієздатності комп'ютерної реалізації розглянемо тестове дослідження, що має рішення, вже одержане засобами теорії ігор.

Наша задача – перевірити якої точності надасть застосування генетичного алгоритму для рішення цієї проблеми

Поставимо завдання. Застосувати адаптивний генетичний алгоритм для розв'язання наступної економічної проблеми:

Торгова фірма розробила кілька варіантів плану продажу товарів на майбутньому ярмарку з урахуванням мінливої кон'юнктури ринку та попиту покупців. Отримані від них можливих поєднань показники доходу представлені у табл. 3.1

Таблиця 3.1 Показники доходу фірми з урахуванням кон'юнктури ринку

	A	B	C	D
1	План продажи	Величина дохода, ден. ед.		
2		$К_1$	$К_2$	$К_3$
3	$П_1$	8	4	2
4	$П_2$	2	8	4
5	$П_3$	1	2	8

Рішення. Переведемо поставлену проблему в термінах задач теорії ігор.

Представимо цю проблему як гру двох гравців: Гравець А – торгова фірма, гравець В – кон'юнктура ринку й попит покупців.

Одержимо математичну модель.

Максимізувати середній прибуток фірми U_A та надати рекомендації по плануванню продажу товарів для населення.

$$U_A = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij}\, p_i q_j \rightarrow max$$

де $S_A = (p_1, p_2, p_3)$; $S_B = (q_1, q_2, q_3)$ – змішані стратегії гравців А і В.

Додаткові умови $p_1 + p_2 + p_3 = 1$, $q_1 + q_2 + q_3 = 1$.

Було проведено ініціалізацію початкового покоління в кількості 100 хромосом й проведено розрахунки, умовою останову генетичного алгоритму задавалось проведення еволюції шляхом зміни заданої кількості поколінь. За результатами досліджень будувався графік залежності Fitness = f (n), де n к-цільк кість поколінб, що змінюється в гашеточному алгоритму (рис. 3.2).

Рисунок 3.2 – Залежність значення Fitness кількості поколінь

Рішення, що одержано становить:

Значення Fitness-функції = 4. 3184.

Значення хромосоми, що привела функцію до оптимума (0,445; 0,25; 0,305), шо представляють змішані стратегії гравця А (p_1, p_2, p_3).

Як вже вказувалося, ця задача має рішення засобами методів, що застосовуються в теорії ігор і рішення її [4] :

$$U_A = 4{,}355555556$$

При змішаних стратегіях S_A = (0,444; 0,245; 0,311)/

Співтавівши результати обчислень робимо висновок, що похибка обчислень має бути 0,01, що є достатньо добре.

Приклад №2.

Змоделюємо ситуацію, що маємо модель біматричної гри, що описується матрицями гравців А і В:

$$\text{Гравець А: } \begin{matrix} 8 & 4 & 2 \\ 2 & 8 & 4 \\ 1 & 2 & 8 \end{matrix} \qquad \text{Гравець В: } \begin{matrix} 7 & 5 & 1 \\ 3 & 7 & 5 \\ 0 & 3 & 7 \end{matrix}$$

Результати розрахунків Fitness-функції в залежності від кількості поколінь надано га графіку (див. рис. 3.3)

Рисунок 3.3 – Залежність значення Fitness кількості поколінь для біматричної гри

В цьому прикладі проведено вже зміну 50 поколінь.

В цьому прикладі вихід Fitness -функції на плато має затяжний характер.

ВИСНОВКИ

В магістерській роботі проведені дослідження на предмет адаптації генетичного алгоритму для рішення біматричних задач теорії ігор. Створено та протестоване інформаційно-програмне забезпечення. Результатами виконання роботи є :

1. Генетичний алгоритм адаптовано для рішення задач теорії ігор.
2. Сформована математична модель теорії ігор та одержано рішення її генетичним алгоритмом.
3. Проведені тестові розрахунки, що надають рішення задач з заданою точністю обчислень.
4. В біматричній грі в порівнянні з антагоністичною, генетичному алгоритму потребується проходження більше поколінь, для отримання заданої точності обчислення.

Дослідження мають практичне застосування та теоретичний інтерес.

СПИСОК ЛІТЕРАТУРИ

1. Математичні методи дослідження операцій: підручник/ Є. А. Лавров, Л. П. Перхун, В. В. Шендрик та ін. – Суми: Сумський державний університет, 2017. – 212 с.
2. Osborn M.J. An Introduction to Game Theory, International Edition International, 2018, Oxford University Press – 556 p.
3. Shagin, V. L. Game Theory: tutorial and workshop / L. V. Shagin. — M.: Urait, 2017. — 223 p.
4. Олешко Т.І. , Лобанов М.О. Економічні задачі в теорії ігор// Проблеми системного підходу в економіці, випуск № 3(65), 2018 с. 120-124.
5. Диксит А. Стратегические игры. Доступный учебник по теории игр / А. Диксит, С. Скит, Д. Рейли-младший. – Москва: Манн, Иванов и Фербер (МИФ), 2017. – 880 с.
6. Сивоконь В.В., Шаповалов С. П. Адаптація чисельних методів до розв’язання задач теорії ігор// Матеріали науково-технічної конференції «Інформатика, математика, автоматика», Суми, СУмДУ, 2020. – с. 53.
7. Карюкин В. В., Чаусов Ф. С. Рефлексивный анализ биматричных игр как решение задачи выбора в моделях противодействия [Електронний ресурс] – Режим доступу до ресурсу: http://www.intelros.ru/pdf/rpu/01_02_2010/07.pdf
8. Бойко Н. І., Михайлишин В. Ю. Ефективність застосування генетичних алгоритмів для пошуку оптимізованих рішень// Інформаційні системи та мережі, 2016, №854. – с. 249-257.
9. Савостян В.В., Шовкопляс О. А., Шаповалов С. П. Адаптація генетичного алгоритму для розв’язання економічних оптимізаційних задач. //Матеріали XII студентської конференції «Перший крок у науку», Суми, СУмДУ, 2021. – с. 174.
10. Game theory// Stanford School of Engineering [Електронний ресурс] – Режим доступу до ресурсу: https://online.stanford.edu/courses/soe-ycs0002-game-theory.

11. Kannan R., Theobald T. GAMES OF FIXED RANK: A HIERARCHY OF BIMATRIX GAMES. [Електронний ресурс] – Режим доступу до ресурсу: https://www.math.uni-frankfurt.de/~theobald/publications/economic.pdf
12. Біматрична гра. Онлайн-калькулятор. [Електронний ресурс] – Режим доступу до ресурсу: https://math.semestr.ru/games/bimatrix.php
13. Шилдт Г. Java. Полное руководство. 10 – издание. Том 1. / Г. Шилдт. – Издательство Диалектика.: Киев, 2020. – 730 с.
14. Шилдт Г. Java. Полное руководство. 10 – издание. Том 2. / Г. Шилдт. – Издательство Диалектика.: Киев, 2020. – 780 с.

ДОДАТОК А

```
package com.company;

import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

/* Антагоністична гра */
public class GeneticAlgorithmZeroSum {
    /* Матриця виграшів */
    public static double[][] valuesMatrix = new double[][]{
            {8, 4, 2},
            {2, 8, 4},
            {1, 2, 8}
    };

    public static void main(String[] args) {
        /* Генеруємо 100 хромосом */
        List<Chromosome> chromosomes = generateChromosomes(100);
        /* Ітеруємося через 30 поколінь */
        for (int i = 0; i < 30; i++) {
            /* Сортуємо хромосоми за значенням фітнес функції */
            chromosomes.sort(Comparator.comparing(Chromosome::getFitness));
            System.out.println("Fitness value: " + chromosomes.get(0).getFitness()
                    + ", P values: " + Arrays.toString(chromosomes.get(0).getGenesP()));
            /* Проводимо кросинговер */
            crossoverPopulation(chromosomes);
```

```
        /* Проводимо мутацію */
        mutatePopulation(chromosomes);
    }
}

/* Генерує популяцію хромосом */
private static List<Chromosome> generateChromosomes(int number) {
    List<Chromosome> chromosomes = new LinkedList<>();
    for (int i = 0; i < number; i++) {
        chromosomes.add(generateChromosome());
    }

    return chromosomes;
}

/* Генерує хромосому */
private static Chromosome generateChromosome() {
    double[] genesP = generateGenes();
    double fitness = calculateFitness(genesP);

    return new Chromosome(genesP, fitness);
}

/* Генерує набір генів хромосоми */
private static double[] generateGenes() {
    double[] genes;
```

```
    do {
        genes = ThreadLocalRandom.current().doubles(3).toArray();
    } while (!validateGeneratedGenes(genes));

    return genes;
}

/* Перевіряє набір генів на відповідність умові a*x1 + b*x2 + c*x3 >= 1 */
private static boolean validateGeneratedGenes(double[] genes) {
    for (int column = 0; column < valuesMatrix[0].length; column++) {
        double sum = 0;
        for (int row = 0; row < valuesMatrix.length; row++) {
            sum += valuesMatrix[row][column] * genes[row];
        }
        if (sum < 1) {
            return false;
        }
    }

    return true;
}

/* Розраховує значення фітнес функції для хромосоми */
private static double calculateFitness(Chromosome chromosome) {
    return calculateFitness(chromosome.getGenesP());
}
```

```
    /* Розраховує значення фітнес функції для набору генів */
    private static double calculateFitness(double[] genesP) {
        return Arrays.stream(genesP).sum();
    }

    /* Проводить кросинговер популяції */
    private static void crossoverPopulation(List<Chromosome> chromosomes) {
        /* Кросинговер проводимо для 60 кращих хромосом, попарно для поточної і наступної хромосом */
        for (int i = 0; i < 60; i += 2) {
            crossoverChromosomes(chromosomes.get(i), chromosomes.get(i + 1));
        }
    }

    /* Проводить кросинговер 2 хромосом */
    private static void crossoverChromosomes(Chromosome firstChromosome, Chromosome secondChromosome) {
        List<double[]> crossoverGenesP = crossoverGenes(firstChromosome.getGenesP(), secondChromosome.getGenesP());
        /* crossoverGenesP.size() != 0 якщо кросинговер можливий для даних 2 хромосом */
        if (crossoverGenesP.size() != 0) {
            firstChromosome.setGenesP(crossoverGenesP.get(0));
            secondChromosome.setGenesP(crossoverGenesP.get(1));
            firstChromosome.setFitness(calculateFitness(firstChromosome));
            secondChromosome.setFitness(calculateFitness(secondChromosome));
```

```
        }
    }

    /* Проводить кросинговер 2 наборів генів */
    private static List<double[]> crossoverGenes(double[] firstGenes, double[] secondGenes) {
        /* Перевіряємо кожен ген і робимо кросинговер для першого можливого або не робимо взагалі */
        for (int i = 0; i < firstGenes.length; i++) {
            /* Обмінюємо ген і */
            double[] firstGenesRes = Arrays.copyOf(firstGenes, firstGenes.length);
            double[] secondGenesRes = Arrays.copyOf(secondGenes, secondGenes.length);
            double temp = firstGenesRes[i];
            firstGenesRes[i] = secondGenesRes[i];
            secondGenesRes[i] = temp;
            /* Перевіряємо чи нові гени валідні */
            if (validateGeneratedGenes(firstGenesRes) && validateGeneratedGenes(secondGenesRes)) {
                firstGenes = Arrays.copyOf(firstGenesRes, firstGenesRes.length);
                secondGenes = Arrays.copyOf(secondGenesRes, secondGenesRes.length);
                return Arrays.asList(firstGenes, secondGenes);
            }
        }

        return Collections.emptyList();
    }
```

```
/* Проводить мутацію останніх 40 хромосом */
private static void mutatePopulation(List<Chromosome> chromosomes) {
    for (int i = 60; i < chromosomes.size(); i++) {
        mutateChromosome(chromosomes.get(i));
    }
}

/* Проводить мутацію хромосоми */
private static void mutateChromosome(Chromosome chromosome) {
    chromosome.setGenesP(mutateGenes(chromosome.getGenesP()));
    chromosome.setFitness(calculateFitness(chromosome));
}

/* Проводить мутацію набору генів */
private static double[] mutateGenes(double[] genes) {
    double newGeneValue;
    Random random = new Random();
    double[] genesRes;
    /* Повторюємо процес поки новий набір генів не стане валідним */
    do {
        genesRes = Arrays.copyOf(genes, genes.length);
        int geneToMutate = getRandomNumber(0, 3);
        newGeneValue = random.nextDouble();
        genesRes[geneToMutate] = newGeneValue;
    } while (!validateGeneratedGenes(genesRes));
```

```
        return genesRes;
    }

    private static int getRandomNumber(int min, int max) {
        Random random = new Random();
        return random.nextInt(max - min) + min;
    }
}
```

ДОДАТОК В

```
package com.company;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

public class GeneticAlgorithmBimatrixGame {
   /* Матриця гравця A */
   public static double[][] valuesAMatrix = new double[][]{
         {8, 4, 2},
         {2, 8, 4},
         {1, 2, 8}
   };

   /* Матриця гравця B */
   public static double[][] valuesBMatrix = new double[][]{
         {7, 5, 1},
         {3, 7, 5},
         {0, 3, 7}
   };

   public static void main(String[] args) {
      /* Генеруємо 100 хромосом */
      List<Chromosome> chromosomes = generateChromosomes(100);
      /* Ітеруємося через 50 поколінь */
      for (int i = 0; i < 50; i++) {

chromosomes.sort(Comparator.comparing(Chromosome::getFitness).reversed());
```

```
        System.out.println("Fitness value: " + chromosomes.get(0).getFitness()
                + ", P values: " + Arrays.toString(chromosomes.get(0).getGenesP())
                + ", K values: " + Arrays.toString(chromosomes.get(0).getGenesK()));
        /* Проводимо кросинговер */
        crossoverPopulation(chromosomes);
        /* Проводимо мутацію */
        mutatePopulation(chromosomes);
    }
}

/* Генерує популяцію хромосом */
private static List<Chromosome> generateChromosomes(int number) {
    List<Chromosome> chromosomes = new LinkedList<>();
    for (int i = 0; i < number; i++) {
        chromosomes.add(generateChromosome());
    }

    return chromosomes;
}

/* Генерує хромосому */
private static Chromosome generateChromosome() {
    double[] genesP = generateGenes();
    double[] genesK = generateGenes();
    double fitness = calculateFitness(genesP, genesK);

    return new Chromosome(genesP, genesK, fitness);
}

/* Генерує набір генів хромосоми */
```

```
    private static double[] generateGenes() {
        double[] randomNumbers = ThreadLocalRandom.current().doubles(3).map(i -> round(i, 3)).toArray();
        double sumRandomNumbers = Arrays.stream(randomNumbers).sum();
        /* Кожне згенероване число ділимо на їх суму щоб отримати числа, які в сумі дадуть 1 */
        return Arrays.stream(randomNumbers).map(i -> round(i / sumRandomNumbers, 3)).toArray();
    }

    /* Розраховує значення фітнес функції для хромосоми */
    private static double calculateFitness(Chromosome chromosome) {
        return calculateFitness(chromosome.getGenesP(), chromosome.getGenesK());
    }

    /* Розраховує значення фітнес функції для наборів генів */
    public static double calculateFitness(double[] genesP, double[] genesK) {
        double fitness = 0;
        for (int i = 0; i < genesP.length; i++) {
            for (int j = 0; j < genesK.length; j++) {
                fitness += GeneticAlgorithmBimatrixGame.valuesAMatrix[i][j] * genesP[i] * genesK[j];
                fitness += GeneticAlgorithmBimatrixGame.valuesBMatrix[i][j] * genesP[i] * genesK[j];
            }
        }

        return fitness;
    }
```

```
    /* Проводить кросинговер популяції */
    private static void crossoverPopulation(List<Chromosome> chromosomes) {
        for (int i = 0; i < 60; i += 2) {
            /* Кросинговер проводимо для 60 кращих хромосом, попарно для поточної і наступної хромосом */
            crossoverChromosomes(chromosomes.get(i), chromosomes.get(i + 1));
        }
    }

    /* Проводить кросинговер 2 хромосом */
    private static void crossoverChromosomes(Chromosome firstChromosome, Chromosome secondChromosome) {
        List<double[]> crossoverGenesP = crossoverGenes(firstChromosome.getGenesP(), secondChromosome.getGenesP());
        /* crossoverGenesP.size() != 0 якщо кросинговер можливий для даних 2 хромосом */
        if (crossoverGenesP.size() != 0) {
            firstChromosome.setGenesP(crossoverGenesP.get(0));
            secondChromosome.setGenesP(crossoverGenesP.get(1));
        }

        /* crossoverGenesP.size() != 0 якщо кросинговер можливий для даних 2 хромосом */
        List<double[]> crossoverGenesK = crossoverGenes(firstChromosome.getGenesK(), secondChromosome.getGenesK());
        if (crossoverGenesK.size() != 0) {
            firstChromosome.setGenesK(crossoverGenesK.get(0));
            secondChromosome.setGenesK(crossoverGenesK.get(1));
        }
```

```
        /* crossoverGenesP.size() != 0 якщо кросинговер можливий для даних 2 хро-
мосом */
        if (crossoverGenesP.size() != 0 || crossoverGenesK.size() != 0) {
            firstChromosome.setFitness(calculateFitness(firstChromosome));
            secondChromosome.setFitness(calculateFitness(secondChromosome));
        }
    }

    /* Проводить кросинговер 2 наборів генів */
    private static List<double[]> crossoverGenes(double[] firstGenes, double[]
secondGenes) {
        /* Індекс i - ген, який будемо обмінювати між наборами */
        for (int i = 0; i < firstGenes.length; i++) {
            /* Індекс j - ген, який будемо фіксвати відносно i для того, щоб вираху-
вати значення останнього */
            for (int j = 0; j < secondGenes.length; j++) {
                if (i != j) {
                    /* Перевіряємо що кросинговер можливий */
                    if (firstGenes[i] + secondGenes[j] < 1 && firstGenes[j] +
secondGenes[i] < 1) {
                        /* Обмінюємо ген i */
                        double temp = firstGenes[i];
                        firstGenes[i] = secondGenes[i];
                        secondGenes[i] = temp;

                        /* Обчислюємо індекс останнього гену */
                        int thirdIndex = 3 - i - j;
                        /* Обчислюємо його значення відносно i та j */
                        firstGenes[thirdIndex] = round(1 - firstGenes[i] - firstGenes[j], 3);
```

```
                secondGenes[thirdIndex] = round(1 - secondGenes[i] -
secondGenes[j], 3);

                return Arrays.asList(firstGenes, secondGenes);
            }
        }
    }
}

    return Collections.emptyList();
}

/* Проводить мутацію останніх 40 хромосом */
private static void mutatePopulation(List<Chromosome> chromosomes) {
    for (int i = 60; i < chromosomes.size(); i++) {
        mutateChromosome(chromosomes.get(i));
    }
}

/* Проводить мутацію хромосоми */
private static void mutateChromosome(Chromosome chromosome) {
    chromosome.setGenesP(mutateGenes(chromosome.getGenesP()));
    chromosome.setGenesK(mutateGenes(chromosome.getGenesK()));
    chromosome.setFitness(calculateFitness(chromosome));
}

/* Проводить мутацію набору генів */
private static double[] mutateGenes(double[] genes) {
    List<Integer> indexes = new LinkedList<>(Arrays.asList(0, 1, 2));
    /* Знаходимо ген, який будемо піддавати мутації */
```

```
    int geneToMutate = indexes.get(getRandomNumber(0, indexes.size()));
    indexes.remove(new Integer(geneToMutate));
    /* Знаходимо ген, значення якого не будемо змінювати */
    int geneToLeaveAsIs = indexes.get(getRandomNumber(0, indexes.size()));
    double newGeneValue;
    /* Генеруємо нове валідне значення гена */
    while (true) {
        Random random = new Random();
        newGeneValue = round(random.nextDouble(), 3);
        if (newGeneValue + genes[geneToLeaveAsIs] < 1) {
            break;
        }
    }
    /* Обчислюємо індекс останнього гену */
    int thirdIndex = 3 - geneToMutate - geneToLeaveAsIs;
    genes[geneToMutate] = newGeneValue;
    /* Обчислюємо його значення відносно мутованого та незмінного гена */
    genes[thirdIndex] = round(1 - newGeneValue - genes[geneToLeaveAsIs], 3);

    return genes;
}

private static double round(double value, int places) {
    if (places < 0) {
        throw new IllegalArgumentException();
    }

    BigDecimal bd = new BigDecimal(Double.toString(value));
    bd = bd.setScale(places, RoundingMode.HALF_UP);
    return bd.doubleValue();
```

```
    }

    private static int getRandomNumber(int min, int max) {
        Random random = new Random();
        return random.nextInt(max - min) + min;
    }
}
```