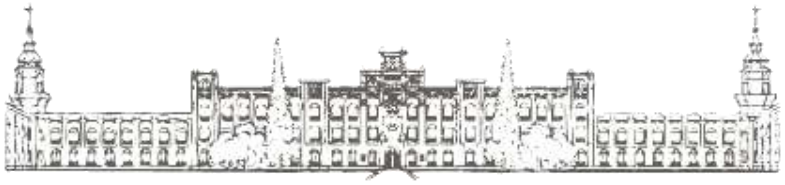


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту

«Технології паралельних обчислень. Курсова робота»

Тема:

Еволюційний (генетичний) алгоритм пошуку оптимального значення

Керівник:
ст. викл. Дифучин Антон Юрійович

«Допущено до захисту»
_____

«___» _____ 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____

_____

Виконавець:
Дейнега Владислав Миколайович
студент групи ІП-13
залікова книжка № ІП-1310
_____

«14» червня 2024 р.

Інна СТЕЦЕНКО

Антон ДИФУЧИН

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Опис алгоритму забезпечити у вигляді псевдокоду. Провести тестування алгоритму та зробити висновок про коректність розробленого алгоритму. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму у відповідності до обраного завдання та обраного програмного забезпечення для реалізації. Опис алгоритму забезпечити у вигляді псевдокоду. Забезпечити ініціалізацію даних при будь-якому великому заданому параметрі кількості даних.
4. Виконати тестування алгоритму, що доводить коректність результатів обчислень. Тестування алгоритму обов’язково проводити на великій кількості даних. Коректність перевіряти порівнянням з результатами послідовного алгоритму.
5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.
6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення не менше 1,2.
7. Дослідити вплив параметрів паралельного алгоритму на отримуване прискорення. Один з таких параметрів – це кількість підзадач, на які поділена задача при розпаралелюванні її виконання.
8. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

АНОТАЦІЯ

Структура та обсяг роботи. Пояснювальна записка курсової роботи складається з 5 розділів, містить 6 рисунків, 2 таблиці, 4 джерела.

Дослідження та реалізація паралельного алгоритму еволюційний (генетичний) алгоритм пошуку оптимального значення мовою Python. Порівняння ефективності паралельної реалізації алгоритму з послідовною.

КЛЮЧОВІ СЛОВА: ЕВОЛЮЦІЙНИЙ АЛГОРИТМ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ЗАДАЧА РЮКАЗАКА

ЗМІСТ

ВСТУП

В теперішній час використання паралельних обчислень стає все більш актуальним та поширеним. Виконання складних алгоритмів та обрахунків вимагає велику кількість часу та ресурсів. Вирішенням цієї проблеми є паралельні технології.

Генетичний алгоритм є одним з найпоширеніших алгоритмів оптимізації, який використовує принципи природного відбору для знаходження наближених рішень складних задач. Головною ідеєю є імітація процесу еволюції поколінь. З кожним новим поколінням алгоритм прагне до поліпшення якості рішень, поступово наближаючись до оптимального або достатньо хорошого рішення. Генетичні алгоритми широко застосовуються в різних областях, включаючи машинне навчання, управління, економіку, та інженерію, завдяки їх здатності ефективно знаходити рішення в складних і великомасштабних просторах пошуку. Але алгоритм має великий недолік, для пошуку оптимального рішення потрібно велика кількість часу на еволюцію поколінь.

В даній курсовій роботі буде розроблений, реалізований та досліджений генетичний алгоритм та його паралельна версія з використанням мови програмування Python.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

1.1 Опис алгоритму

Генетичний алгоритм[1] - це метод евристичного пошуку, натхненний біологічними процесами еволюції. Він використовується для вирішення складних задач оптимізації, де знайти оптимальне або близьке до оптимального рішення традиційними методами може бути складно або неможливо.

Головні етапи алгоритму такі:

- Створення первинної популяції. Алгоритм створює первинний набір хромосом, тобто набір параметрів для пошуку оптимального рішення.
- Відбір батьків. В ході «еволюції» відбувається відбір батьків для майбутніх нащадків. Є багато варіацій відбору, від випадкового, до змагання групи хромосом.
- Схрещування батьків. Після відбору відбувається схрещування відібраних хромосом, для створення нової хромосоми з генами двох батьків.

Цей цикл повторюється до тих пір, поки не буде знайдено оптимальне або близьке до оптимального рішення.

Генетичний алгоритм використовується в багатьох сферах, наприклад, пошук оптимальних маршрутів. В сфері фінансів алгоритм може допомогти розробити бізнес стратегію саме завдяки можливості гнучко застосовуватись до різноманітних задач.

1.2 Готові рішення

Паралельна реалізація генетичного алгоритму дозволяє значно зменшити час пошуку оптимального рішення. Особливо корисно це під час вирішення складних задач, які вимагають великої кількості обчислень.

Існує два найбільш поширених підходи до паралельної реалізації генетичного алгоритму.

Перший метод, це розподіл популяції[2], або ж острівний метод. Наглядну схему алгоритму наведена на рисунку 1.1. Головною ідеєю цього методу є розподіл популяції на декілька частин, які будуть обчислювати в різних потоках. Кожна частина еволюціонує незалежно протягом кількох поколінь. Після деякої кількості ітерацій, потоки обмінюються найкращими хромосомами, що дозволяє зберегти генетичне різноманіття та уникнути локальних мінімумів. Після обміну популяції знову розподіляються на потоки, і процес продовжується. Ці кроки повторюються допоки не буде знайдене оптимальне рішення.

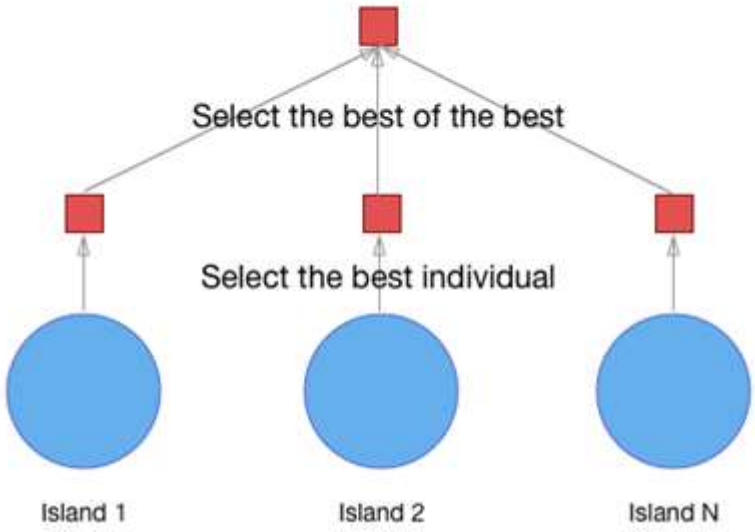


Рисунок 1.1 – Схема паралельного алгоритму острівного методу[3]

Другий метод, це розподіл обчислень, де різні стадії генетичного алгоритму виконуються в окремих потоках. Наприклад виконання відбору, схрещування або мутації в окремих потоках. Це дозволяє більш ефективно використовувати багатоядерні процесори, оскільки кожна з цих операцій може бути досить вимогливою до ресурсів. Окремі етапи генетичного алгоритму можуть виконуватися одночасно, без необхідності чекати завершення попередніх етапів, що допомагає зменшити загальний час обчислень,.

Таким чином, обидва підходи – розподіл популяції та розподіл обчислень дозволяють суттєво прискорити виконання генетичних алгоритмів, роблячи їх більш придатними для розв'язання складних задач, особливо для задач, які

вимагають великої кількості розрахунків. Існують багато варіантів паралельного алгоритму, а тут були наведені лише дві головні.

1.3 Практичне використання програмами

Для практичного сенсу роботи програми було поставлено мету зробити максимально можливий різноманітний вибір на прикладі формуванню комплекту настільних ігор індивідуально для кожного запуску програми, тобто алгоритм зіставляє, так звані, «випадкові» рекомендації з урахуванням найбільшої вигоди для клубу настільних ігор.

Ціллю розробленої програми є задоволення кожного клієнта клубу настільних ігор шляхом збирання комплекту ігор більш різноманітного жанру та більш вигідним для клубу.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Розробка

Задачею для вирішення, було обрано модифіковану задачу рюкзака. Модифікація представляє собою додаткові параметри, які враховуються під час підрахунку пристосованості. Першим критерієм є популярність, тобто наскільки предмет потрібен. Другий параметр це жанр предмету. Саме різноманітність жанрів враховується окрім ціни (ваги) предмета та його популярності.

Реальним прикладом, де може використовуватись такий алгоритм, це комплектація набору настільних ігор. Алгоритм допоможе врахувати популярність та різноманіття настільних ігор, з обмеженням в бюджеті.

Програма складається з таких класів:

ClassicGenetic – це клас, який реалізує класичний генетичний алгоритм.

Метод wheel_selection() реалізує механізм рулетки для вибору хромосом на основі їх фітнес-значень.

Метод crossover() приймає об’єкти двох батьків для подальшого схрещування. Кросовер (схрещування) відбувається між двома батьківськими хромосомами для створення нової дитини. Гени дитини вибираються випадково від одного з батьків. На вибір генів батьків впливає пристосованість, чим більша пристосованість одного з батьків, тим більше шанс передати ген. Метод виконує мутацію на хромосомі з імовірністю mutationRate.

Методі mutate() приймає об’єкт хромосоми для подальшої мутації. Мутація відбувається шляхом випадкової зміни трьох генів у хромосомі.

Метод genetic_algorithm – це головний метод класу. В ньому реалізований один цикл генетичного алгоритму.

Population – це клас, який реалізує популяцію. Популяція складається з хромосом, які в свою чергу складаються з генів.

Chromosome – це клас, який реалізує хромосоми. Він зберігає пристосованість окремої хромосоми та зберігає набір генів, який представлений класом **Gene.** Кожен ген має свою вартість, популярність та жанр. Набір генів

представлений у вигляді послідовності з 1 та 0, де 1 означає, що ген включено в дану хромосому, а 0 – що ген не включено. Пристосованість хромосоми обраховується в методі calc_fitness().

Метод

2.2 Псевдокод

Псевдокод основної частини алгоритму

```
Function genetic_algorithm
    Initialize temp_population with top chromosomes from current population
    For i in range(elitism, population_size):
        parent1 = wheel_selection(current_population)
        parent2 = wheel_selection(current_population)
        child = crossover(parent1, parent2)
        child = mutate(child)
        Append child to temp_population
    result_population = Create new population from temp_population
    Evaluate fitness of result_population
    Update current population with result_population
    Return result_population
```

Псевдокод мутації

```
Function mutate(Chromosome chromosome):
    If random < mutation_rate:
        For 3 times:
            mutation_point = random_int(0, length(chromosome.genes) - 1)
            If chromosome.genes[mutation_point] == "1":
                chromosome.genes[mutation_point] = "0"
            Else:
                chromosome.genes[mutation_point] = "1"
    Return chromosome
```

Псевдокод схрещування

```
Function crossover(Chromosome parent1, Chromosome parent2):
```

```
    child = Initialize empty chromosome
    For i in range(size of parent1):
        rand = random_int(0, abs(int(parent1.fitness + parent2.fitness)))
        If rand <= parent1.fitness:
            Append parent1.genes[i] to child
        Else:
            Append parent2.genes[i] to child
    Return child
```

Псевдокод відбору

```
Function wheel_selection(Population P):
    total_fitness = Sum(abs(fitness) for each chromosome in P)
    For each chromosome in P:
        rand = random_uniform(0, total_fitness)
        current_sum = 0
        For each chromosome in P:
            current_sum += abs(chromosome.fitness)
            If current_sum >= rand:
                Return chromosome
```

2.3 Аналіз швидкодії

Базовий алгоритм був протестований при різних значеннях величини популяції. В таблиці 2.1 можна побачити результат тестування, а саме таблицю залежності часу виконання від розміру популяції в мілісекундах.

З отриманих даних ми можемо побачити, що час виконання сильно зростає зі збільшенням кількості хромосом.

Таблиця 2.1 – результат тестування послідовного алгоритму

Кількість хромосом	Час виконання
100	13,55
500	100,54
1000	270,80
2500	1318,27
5000	4655,11
7500	9378,95
10000	15501,36

3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

Для реалізації паралельного генетичного алгоритму була обрана мова програмування Python. Python є високорівневою мовою програмування, яка має чіткій та зрозумілій синтаксис.

Переваги використання Python для паралельних алгоритмів:

- Бібліотека multiprocessing: дозволяє створювати окремі процеси, які можуть виконувати завдання паралельно, що допомагає обійти обмеження GIL у Python.
- Також Python має багато бібліотек які відкривають можливості до написання функціонального, ефективного та зрозумілого коду.
- Python працює на різних операційних системах без потреби вносити зміни в код, що робить його ідеальним для кросплатформенних розробок.

Недоліки використання Python для паралельних алгоритмів:

- GIL є однією з основних проблем багатопотоковості в Python. Він дозволяє лише одному потоку виконуватись у будь-який момент часу, що може бути вузьким місцем для CPU-bound задач. На щастя, існує бібліотека multiprocessing, яка дозволяє обійти це обмеження.
- Хоча Python є інтерпретованою мовою і має дещо нижчу продуктивність порівняно з компільованими мовами.
- Використання окремих процесів для паралелізму призводить до високих витрат на пам'ять, бо кожен процес має власний простір пам'ять.

У висновку, Python є чудовим вибором для реалізації паралельних генетичних алгоритмів завдяки своїй простоті використання, багатому набору бібліотек та можливості працювати на різних платформах. Незважаючи на певні обмеження, такі як GIL та продуктивність, правильне використання бібліотек, таких як multiprocessing, дозволяє ефективно вирішувати задачі оптимізації.

4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

4.1 Проектування алгоритму

Перед реалізацією паралельного алгоритму було досліджені можливі шляхи використання паралельних технологій. З можливих варіантів було виявлено два:

Паралельно виконувати обрахунок пристосованості для кожної особини. При складній формулі обрахунку пристосованості це може дати значне прискорення. Але використана формула для обрахунку занадто проста і паралельний обрахунок пристосованості не буде достатньо ефективним.

Паралельно виконувати генерації нащадків. В цей процес входить відбір батьків, схрещування та мутація. Разом ці дії займають багато часу, особливо схрещування батьків. Також цей процес є незалежним, через що не буде витрачатись час та ресурси на синхронізацію. Тому було обрано паралельно виконувати створення нащадків. Схема взаємодії потоків наведено в Додатку В.

Для реалізації паралельного алгоритму було використано бібліотеку multiprocessing, а саме multiprocessing.Pool. Однією з основних переваг multiprocessing є те, що вона дозволяє обійти Global Interpreter Lock (GIL), який є обмеженням в стандартній реалізації Python (CPython). GIL унеможливлює одночасне виконання багатьох потоків Python коду через блокування доступу до структур даних на рівні інтерпретатора. Дана бібліотека дозволяє обійти обмеження GIL, та виконувати задачі асинхронно.

map(func, iterable, chunksize=None): Розподіляє ітерацію на кілька процесів та повертає список результатів. Параметр func функція, яка буде виконуватись процесом. Параметр iterable ітераційний об’єкт, такі як списки або інші об’єкти. Парметр chunksize задає розмір чанку. Якщо ж chunksize не вказаний явно, піл автоматично визначить найбільш оптимальний розмір чанків.

Псевдокод алгоритму

@staticmethod

```
        def offspring_task(args):
            obj = args
            parent1 = obj.wheel_selection()
            parent2 = obj.wheel_selection()

            child = obj.crossover(parent1, parent2)
            child = obj.mutate(child)

            child.calc_fitness(obj.population.maxBudget)

            return child

        def genetic_algorithm(self, process_count):
            temp_population = []
            temp_population[:self.elitism] = self.population.get_best_chromosome(self.elitism)

            with multiprocessing.Pool(process_count) as pool:
                offsprings = pool.map(ParallelGenetic.offspring_task, (self for _ in self.population.chromosomes[self.elitism:]))

            temp_population[self.elitism:] = offsprings

            result = Population(self.population.populationSize, self.population.maxBudget, 0, self.population.data)
            result.chromosomes = temp_population

            self.population = result

            return result 4.2 Реалізація
```

Для реалізації паралельного алгоритму, створення нащадків було відокремлене в окремий статичний метод

```
@staticmethod
    def offspring_task(args):
        obj = args
        parent1 = obj.wheel_selection()
        parent2 = obj.wheel_selection()

        child = obj.crossover(parent1, parent2)
        child = obj.mutate(child)

        child.calc_fitness(obj.population.maxBudget)

        return child
```

В нього передається об’єкт класу ParallelGenetic. Далі виконується створення нового нащадка, а саме вибір батьків методом wheel selection, їх схрещування, мутація та підрахунок фітнесу.

Далі створюється пул потоків, в який передається метод та генератором об’єктів створюється масив параметрів

```
with multiprocessing.Pool(process_count) as pool:
    offsprings = pool.map(ParallelGenetic.offspring_task, (self for _ in self.population.chromosomes[self.elitism:]))
```

Оскільки я не вказую явно розмір чанку, то пул автоматично розділяє дані найбільш оптимальним чином.

4.3 Тестування

На рисунках 4.1 ми можемо побачити результати виконання алгоритму. Задача для оптимізації є підбір оптимального набору настільних ігор, для наповнення клубу. Були використані такі параметри:

- Кількість хромосом - 10000

- Кількість предметів - 50
- Відсоток мутації - 0.1
- Елітизм дорівнював 5% від розміру популяції
- Більше результатів надані в додатку В

```
Result
Fitness 155.25565
Popularity 157
Cost 22653
Genres 11
Gene3: 5 1110  Economic
Gene4: 9 920  Puzzle
Gene6: 4 650  Family
Gene7: 8 1555  Science Fiction
Gene8: 6 860  Classic
Gene9: 9 1350  Strategy
Gene11: 6 620  Sci-Fi
Gene13: 4 519  Puzzle
Gene15: 8 999  Classic
Gene17: 5 750  Card
Gene18: 6 1220  Military
Gene20: 6 780  Detective
Gene21: 8 1425  Fantasy
Gene23: 9 940  Puzzle
Gene30: 6 640  Sci-Fi
Gene34: 8 1010  Classic
Gene39: 6 805  Detective
Gene40: 8 1395  Fantasy
Gene42: 9 915  Puzzle
Gene44: 4 670  Family
Gene45: 8 1565  Science Fiction
Gene47: 9 1345  Strategy
Gene49: 6 610  Sci-Fi


Parallel genetic algorithm executed in 11171.06 milliseconds
```

Рисунок 4.1 -результат виконання

Ми можемо побачити що алгоритм надає рішення наближене до оптимально, з можливістю відхилення за максимальний бюджет, оскільки це може бути кращим рішенням.

4.4 Тестування коректності

Для перевірки коректності роботи алгоритму я використав бібліотеку knapsack 0.1.1 [4]. Дана бібліотека є пакетом для вирішення класичної проблеми рюкзака. Результати роботи цієї бібліотеки наведині на рисунку 4.2. Данна бібліотека повертає як результат кортеж, де перше значення це загальна популярність, а друге значення це масив предметів.

```
(153, [4, 8, 9, 10, 11, 13, 15, 20, 23, 28, 29, 30, 32, 34, 36, 39, 42, 44, 46, 47, 48, 49])
```

Рисунок 4.2 – результати роботи бібліотеки

Результати роботи моєї програми при відповідних вхідних даних наведені на рисунку 4.3

```
Fitness 153.3147
Popularity 132
Cost 20089
Genres 11
Gene1: 6 799  Detective
Gene2: 8 1380  Fantasy
Gene4: 9 920  Puzzle
Gene7: 8 1555  Science Fiction
Gene10: 8 1199  Strategy
Gene11: 6 620  Sci-Fi
Gene13: 4 519  Puzzle
Gene14: 7 1518  Cooperative
Gene15: 8 999  Classic
Gene18: 6 1220  Military
Gene21: 8 1425  Fantasy
Gene22: 5 1145  Economic
Gene23: 9 940  Puzzle
Gene32: 4 505  Puzzle
Gene37: 6 1205  Military
Gene39: 6 805  Detective
Gene44: 4 670  Family
Gene46: 6 870  Classic
Gene48: 8 1185  Strategy
Gene49: 6 610  Sci-Fi
```

Рисунок 4.3 – результат виконання

Як ми бачимо результати є досить схожими, але деякі відмінності, оскільки моя реалізація алгоритму враховує не тільки популярність, а і різноманітність

комплекта ігор. Тобто мій алгортм більш оптимальний для бізнесу, оскільки він укомплектовує найбільш вигідні рекомендації, на прикладі клубу настільних, де рекомендуються найбільш вигідні та різноманітні ігри в комплекті.

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Експериментальне дослідження проводилось на системі з такими характеристиками:

Процесор: AMD Ryzen 5 1600 Six-Core Processor(6 ядер, 12 логічних процесорів)

Оперативна пам'ять: 16 ГБ

Операційна система: Windows 10

Тестування проводилось для таких параметрів:

- Кількість хромосом були 100, 500, 1000, 2500, 5000, 10000
- Кількість предметів 50
- Відсоток мутації дорівнював 0.1
- Кількість процесів - 6
- Елітизм дорівнював 5% від розміру популяції

Перед головними замірами був проведений розгін, а саме було виконано 20 запусків паралельного алгоритму.

Результати замірів часу виконання та прискорення представлені у таблиці 5.1

Таблиця 5.1. – Заміри та прискорення

Кількість хромосом	Час послідовного алгоритму, мс	Час паралельного алгоритму, мс	Прискорення
100	14	389	0,03599
500	106	452	0,234513
1000	296	613	0,482871
2500	1409	1504	0,936835
5000	5033	3622	1,389564
7500	11514	6207	1,855002
10000	22864	11171	2,046728

На рисунку 5.1 відображена зрівняння часу виконання паралельного та класичного алгоритмів.

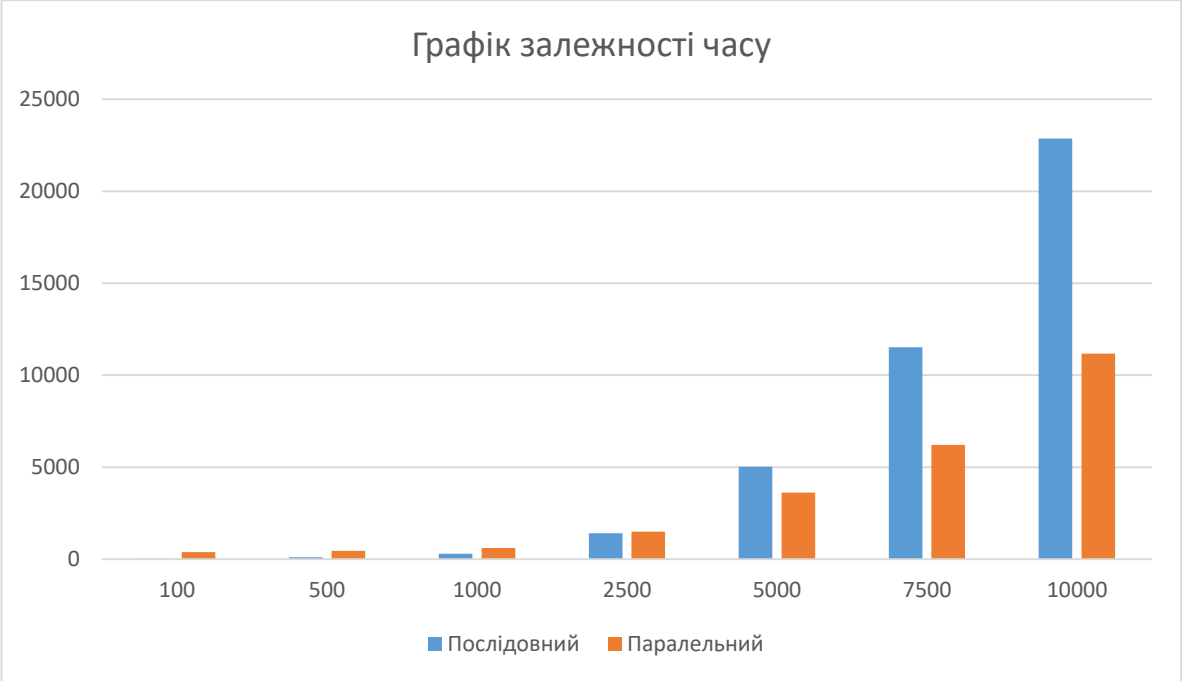


Рисунок 5.1. – Графік зрівняння часу виконання паралельного та класичного алгоритмів

В результаті, ми можемо побачити низьку ефективність паралельного алгоритму на малих кількостях даних. Але зі збільшенням кількості хромосом алгоритм дуже добре себе показує, на противагу класичному алгоритму.

Було досягнуто прискорення до 2.04 при великих розмірах популяції. Ефективне прискорення алгоритму починається після 5000 хромосом в популяції.

Причиною низької ефективності при малих розмірах популяціє є витрати на створення та управління процесами. Час, витрачений на паралелізацію, може перевищувати вигоду від одночасного виконання кількох задач, що призводить до низького прискорення

ВИСНОВКИ

Під час виконання даної курсової роботи, було розроблено та досліджено паралельну реалізацію генетичного алгоритму для вирішення модифікованої задачі рюкзака. Головне завдання дослідження це порівняння паралельного та послідовного алгоритмів на ефективність.

Тестування проводилось для таких параметрів:

- Кількість хромосом були 100, 500, 1000, 2500, 5000, 10000
- Кількість предметів 50
- Відсоток мутації дорівнював 0.1
- Кількість процесів - 6
- Елітизм дорівнював 5% від розміру популяції

За результатами експериментів було виявлена ефективність паралельного алгоритму на розмірах популяції більше 5000 та прискорення яке досягало до 2.04

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Brital A. Genetic algorithm explained:. Medium. URL: https://medium.com/@AnasBrital98/genetic-algorithm-explained-76dfbc5de85d (date of access: 14.06.2024).
2. Cantú-Paz E. Efficient parallel genetic algorithms: theory and practice. Sciencedirect. URL: https://www.sciencedirect.com/science/article/abs/pii/S0045782599003850 (date of access: 14.06.2024).
3. Poddar S. Parallel genetic algorithm. Medium. URL: https://medium.com/swlh/parallel-genetic-algorithm-3d3314c8373c (date of access: 14.06.2024).
4. Knapsack. *PyPI*. URL: https://pypi.org/project/knapsack/ (дата звернення: 26.06.2024).

ДОДАТКИ

Додаток А. Лістинг коду

Клас Main

```
from Gene import Gene
from ClassicGenetic import ClassicGenetic
from ParallelGenetic import ParallelGenetic
from Population import Population
from Chromosome import Chromosome
import time
import knapsack


def run_parallel(chromosome_count, generation_count):
    timeList = []
    for i in range(20):
        stop_iter = 0
        population = Population(chromosome_count, 20000, 1, purchases)
        population.fitness_all()
        parallelGenetic = ParallelGenetic(population, 0.3, int(chromosome_count
* 0.05))
        start_time = time.time()
        while stop_iter < generation_count:
            parallelGenetic.genetic_algorithm(6)

            stop_iter += 1
        end_time = time.time()

        total_time_in_seconds = end_time - start_time
        total_time_in_milliseconds = total_time_in_seconds * 1000
        timeList.append(total_time_in_milliseconds)
```

```
            print("Result")
            max_chrom = parallelGenetic.get_max_fitness()
            max_chrom.print_test()
            print()
        mean_time = sum(timeList) / len(timeList)
        print(f"Parallel genetic algorithm executed in {mean_time:.2f} milliseconds")
        for j in range(timeList.__len__()):
            print(f"Time {j + 1}: {timeList[j]:.2f} milliseconds")




    def run_classic(chromosome_count, generation_count):
        stop_iter = 0
        timeList = []
        population = Population(chromosome_count, 20000, 1, purchases)
        population.fitness_all()
        while stop_iter < generation_count:
            classicAlgorithm = ClassicGenetic(population, 0.3, int(chromosome_count
* 0.05))

            start_time = time.time()

            classicAlgorithm.genetic_algorithm()

            end_time = time.time()

            total_time_in_seconds = end_time - start_time
            total_time_in_milliseconds = total_time_in_seconds * 1000
            timeList.append(total_time_in_milliseconds)
```

```
            stop_iter += 1

        print("Result")
        max_chrom = classicAlgorithm.get_max_fitness()
        max_chrom.print_test()
        print()
        mean_time = sum(timeList) / len(timeList)
        print(f"Classic genetic algorithm executed in {mean_time:.2f} milliseconds")
        for j in range(timeList.__len__()):
            print(f"Time {j + 1}: {timeList[j]:.2f} milliseconds")


    if __name__ == '__main__':
        purchases = []
        test_popul = []
        test_cost = []
        with open("./data/chromosomes.txt", 'r') as file:
            for line in file:
                data = line.strip().split(',')

                test_cost.append(int(data[0]))
                test_popul.append(int(data[1]))

                purchase = Gene(int(data[0]), int(data[1]), data[2])
                purchases.append(purchase)



        chromosome_count = [100, 100, 500, 1000, 2500, 5000]
        process_cout = [2, 4, 6, 8, 10, 12]
```

```
        for i in range(len(chromosome_count)):
            print("--------------------------------")
            print(f"Chromosome count: {chromosome_count[i]}")
            print("--------------------------------")

            #run_classic(chromosome_count[i], 20)

            print(knapsack.knapsack(test_cost, test_popul).solve(20000))

            print()

            #Розгін
            for i in range(0, 20):
                i = 0
                stop_iter = 0
                max_fitness = 0
                timeList = []
                population_tpm = Population(chromosome_count[i], 10000, 1,
purchases)
                population_tpm.fitness_all()
                while stop_iter < 20:
                    population = population_tpm
                    parallelGenetic = ParallelGenetic(population, 0.3, int(2500 * 0.05))

                    start_time = time.time()

                    parallelGenetic.genetic_algorithm(6)

                    end_time = time.time()
```

```
                total_time_in_seconds = end_time - start_time
                total_time_in_milliseconds = total_time_in_seconds * 1000
                timeList.append(total_time_in_milliseconds)
                stop_iter += 1

            run_parallel(chromosome_count[i], 20)
```

Клас ClassicGenetic

```
import random
from Chromosome import Chromosome
from Population import Population


class ClassicGenetic:
    def __init__(self, population, mutation_rate, elitism):
        self.population = population
        self.mutationRate = mutation_rate
        self.elitism = elitism

    def print(self):
        self.population.print()

    def get_max_fitness(self):
        return self.population.calc_max_fitness()

    def wheel_selection(self):
        total_fitness = sum(abs(chromosome.fitness) for chromosome in
self.population.chromosomes)
        for _ in range(len(self.population.chromosomes)):
            rand = random.uniform(0, total_fitness)
```

```
            current_sum = 0
            for chromosome in self.population.chromosomes:
                current_sum += abs(chromosome.fitness)
                if current_sum >= rand:
                    return chromosome

        def crossover(self, parent1, parent2):
            child = Chromosome(self.population.chromosomeSize, self.population.data)
            temp = []

            for i in range(parent1.get_size()):
                rand = random.randint(0, abs(int(parent1.get_fitness() + parent2.get_fitness())))
                if rand <= parent1.get_fitness():
                    temp.append(parent1.get_genes()[i])
                else:
                    temp.append(parent2.get_genes()[i])

            child.chromosomes = temp

            return child

        def mutate(self, chromosome):
            if random.random() < self.mutationRate:
                for i in range(3):
                    mutation_point = random.randint(0, len(chromosome.genes) - 1)
                    chromosome.genes[mutation_point] = "0" if chromosome.genes[mutation_point] == "1" else "1"
                return chromosome
```

```
            else:
                return chromosome

        def genetic_algorithm(self):
            temp_population = []
            temp_population[:self.elitism] = self.population.get_best_chromosome(self.elitism)
            for i in range(self.elitism, self.population.populationSize):
                parent1 = self.wheel_selection()
                parent2 = self.wheel_selection()

                child = self.crossover(parent1, parent2)

                child = self.mutate(child)

                temp_population.append(child)

            result = Population(self.population.populationSize, self.population.maxBudget, 0, self.population.data)
            result.chromosomes = temp_population

            self.population = result
            self.population.fitness_all()

            return result
```

Клас Chromosome

```
import numpy as np
from collections import Counter

class Chromosome:
```

```
def __init__(self, size, data):
    self.genes = []
    self.size = size
    self.fitness = 0
    self.data = data

    self.total_popularity = 0
    self.total_cost = 0
    self.unique_genres = 0

    self.genes = np.random.randint(2, size=self.size)
    unique_genres = set()
    for gene in data:
        unique_genres.add(gene.get_genre())
    self.genre_count = len(unique_genres)
    self.popularity_count = sum(abs(gene.popularity) for gene in self.data)

def get_fitness(self):
    return self.fitness

def get_genes(self):
    return self.genes

def get_size(self):
    return self.size

def calc_fitness(self, max_budget):
    self.fitness = 0
    total_cost = 0
    total_count = 0
```

```
        total_popularity = 0
        genre_list = []

        for i in range(self.size):
            if self.genes[i] == 1:
                total_count += 1
                total_popularity += self.data[i].get_popularity()
                total_cost += self.data[i].get_cost()
                genre_list.append(self.data[i].get_genre())

        unique_genres = len(set(genre_list))
        A = 1
        B = 2
        self.fitness = total_popularity * A + unique_genres * B
        if total_cost > max_budget:
            if total_cost >= max_budget*2:
                self.fitness = 0
            else:
                percentage_excess = (total_cost - max_budget) / max_budget * 100
                self.fitness -= self.fitness * (percentage_excess / 100)

        #test
        self.total_popularity = total_popularity
        self.total_cost = total_cost
        self.unique_genres = unique_genres
        #end test

        return self.fitness

    def print_chromosome(self):
```

```
            for i in range(self.size):
                print("Gene" + str(i) + ": " + str(self.data[i].get_popularity()) + " " +
str(self.data[i].get_cost()) + " " + self.data[i].get_genre())

        def print_test(self):
            print("Fitness " + str(self.fitness))
            print("Popularity " + str(self.total_popularity))
            print("Cost " + str(self.total_cost))
            print("Genres " + str(self.unique_genres))
            for i in range(self.size):
                if self.genes[i] == 1:
                    print("Gene" + str(i) + ": " + str(self.data[i].get_popularity()) + " " +
str(self.data[i].get_cost()) + " " + self.data[i].get_genre())
            print()
```

Клас ParallelGenetic

```
import random
import multiprocessing
from Chromosome import Chromosome
from Population import Population

class ParallelGenetic:
    def __init__(self, population, mutation_rate, elitism):
        self.population = population
        self.mutationRate = mutation_rate
        self.elitism = elitism

    def print(self):
        self.population.print()

    def get_max_fitness(self):
```

```
        return self.population.calc_max_fitness()

    def wheel_selection(self):
        total_fitness = sum(abs(chromosome.fitness) for chromosome in self.population.chromosomes)
        for _ in range(len(self.population.chromosomes)):
            rand = random.uniform(0, total_fitness)
            current_sum = 0
            for chromosome in self.population.chromosomes:
                current_sum += abs(chromosome.fitness)
                if current_sum >= rand:
                    return chromosome

    def crossover(self, parent1, parent2):
        child = Chromosome(self.population.chromosomeSize, self.population.data)
        temp = []

        for i in range(parent1.get_size()):
            rand = random.randint(0, abs(int(parent1.get_fitness() + parent2.get_fitness())))
            if rand <= parent1.get_fitness():
                temp.append(parent1.get_genes()[i])
            else:
                temp.append(parent2.get_genes()[i])

        child.chromosomes = temp

        return child
```

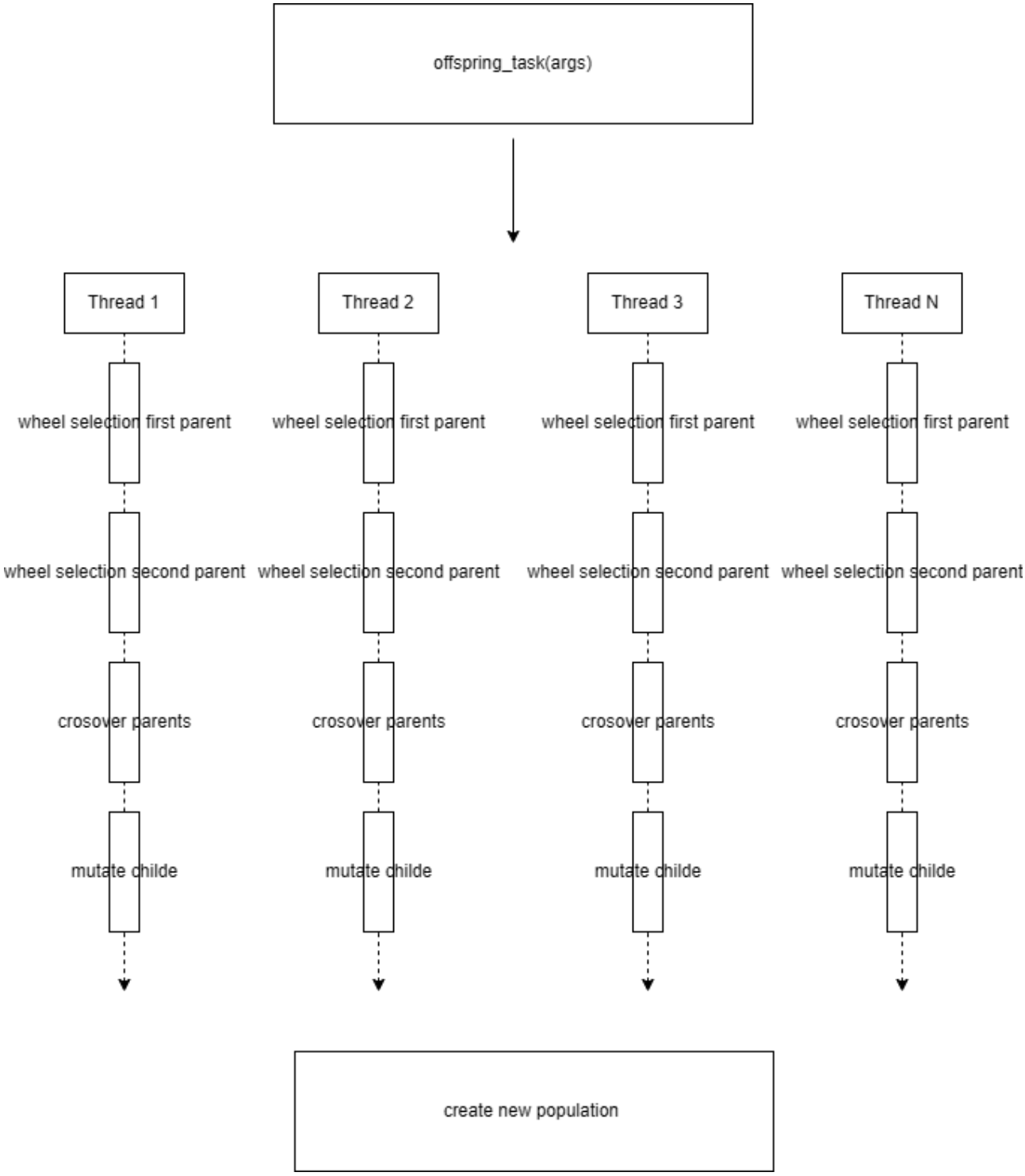
```
        def mutate(self, chromosome):
            if random.random() < self.mutationRate:
                for i in range(3):
                    mutation_point = random.randint(0, len(chromosome.genes) - 1)
                    chromosome.genes[mutation_point] = "0" if
chromosome.genes[mutation_point] == "1" else "1"
                return chromosome
            else:
                return chromosome

        @staticmethod
        def offspring_task(args):
            obj = args
            parent1 = obj.wheel_selection()
            parent2 = obj.wheel_selection()

            child = obj.crossover(parent1, parent2)
            child = obj.mutate(child)

            child.calc_fitness(obj.population.maxBudget)

            return child

        def genetic_algorithm(self, process_count):
            temp_population = []
            temp_population[:self.elitism] =
self.population.get_best_chromosome(self.elitism)

            with multiprocessing.Pool(process_count) as pool:
```

```
            offsprings = pool.map(ParallelGenetic.offspring_task, (self for _ in self.population.chromosomes[self.elitism:]))

            temp_population[self.elitism:] = offsprings

            result = Population(self.population.populationSize, self.population.maxBudget, 0, self.population.data)
            result.chromosomes = temp_population

            self.population = result

            return resultКлас Population
        from Chromosome import Chromosome

        class Population:
            def __init__(self, population_size, max_budget, generate, data):
                self.populationSize = population_size
                self.chromosomeSize = len(data)
                self.maxBudget = max_budget
                self.chromosomes = []
                self.data = data

                if generate == 1:
                    for i in range(population_size):
                        chromosome = Chromosome(self.chromosomeSize, data)
                        self.chromosomes.append(chromosome)

            def fitness_all(self):
                for chromosome in self.chromosomes:
                    chromosome.calc_fitness(self.maxBudget)
```

```
def print(self):
    for i in range(self.populationSize):
        print("Chromosome %d" % i)
        self.chromosomes[i].print_chromosome()
        print()

def print_fitness(self):
    for i in range(self.populationSize):
        print(self.chromosomes[i].get_fitness(), end=" ")

def info(self):
    print("Population size: %d" % self.populationSize)
    print("Max budget: %d" % self.maxBudget)
    print("Chromosome size: %d" % self.chromosomeSize)
    for i in range(self.populationSize):
        print("Chromosome %d" % i)
        print(str(self.chromosomes[i].fitness))

def calc_max_fitness(self):
    max_fitness = self.chromosomes[0].fitness
    temp = self.chromosomes[0]
    for chromosome in self.chromosomes:
        if chromosome.fitness > max_fitness:
            max_fitness = chromosome.fitness
            temp = chromosome
    return temp

def get_best_chromosome(self, elitism):
```

```
            self.chromosomes.sort(key=lambda chromosome: chromosome.fitness,
reverse=True)
            return self.chromosomes[:elitism]
```

Додаток Б. Схема виконання паралельних процесів



Додаток В. Приклади виконання програми

На рисунках 1 - наведено приклади виконання алгоритму

```
Result
Fitness 73.3465
Popularity 65
Cost 11371
Genres 10
Gene5: 7 2000  Mystic
Gene7: 8 1555  Science Fiction
Gene8: 6 860  Classic
Gene10: 8 1199  Strategy
Gene11: 6 620  Sci-Fi
Gene17: 5 750  Card
Gene20: 6 780  Detective
Gene33: 7 1542  Cooperative
Gene40: 8 1395  Fantasy
Gene44: 4 670  Family


Parallel genetic algorithm executed in 439.20 milliseconds
```

Рисунок 1

```
Result
Fitness 60.8832
Popularity 78
Cost 13658
Genres 9
Gene1: 6 799  Detective
Gene2: 8 1380  Fantasy
Gene4: 9 920  Puzzle
Gene5: 7 2000  Mystic
Gene6: 4 650  Family
Gene10: 8 1199  Strategy
Gene16: 9 2200  Role-playing
Gene36: 5 725  Card
Gene37: 6 1205  Military
Gene40: 8 1395  Fantasy
Gene48: 8 1185  Strategy


Parallel genetic algorithm executed in 449.67 milliseconds
```

Рисунок 2

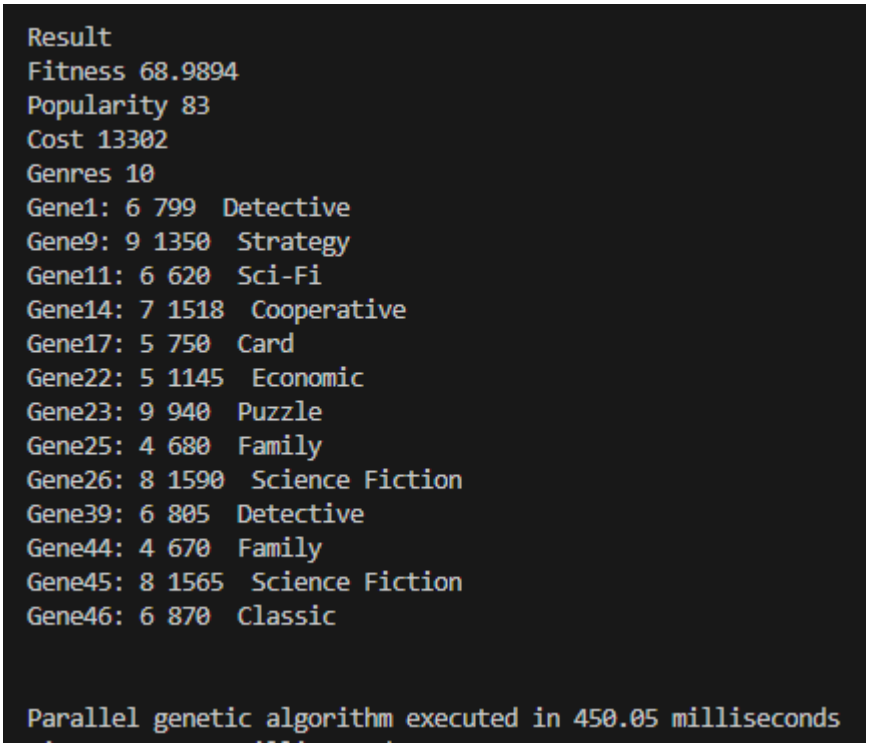

Result
Fitness 68.9894
Popularity 83
Cost 13302
Genres 10
Gene1: 6 799 Detective
Gene9: 9 1350 Strategy
Gene11: 6 620 Sci-Fi
Gene14: 7 1518 Cooperative
Gene17: 5 750 Card
Gene22: 5 1145 Economic
Gene23: 9 940 Puzzle
Gene25: 4 680 Family
Gene26: 8 1590 Science Fiction
Gene39: 6 805 Detective
Gene44: 4 670 Family
Gene45: 8 1565 Science Fiction
Gene46: 6 870 Classic
Parallel genetic algorithm executed in 450.05 milliseconds

Рисунок 3