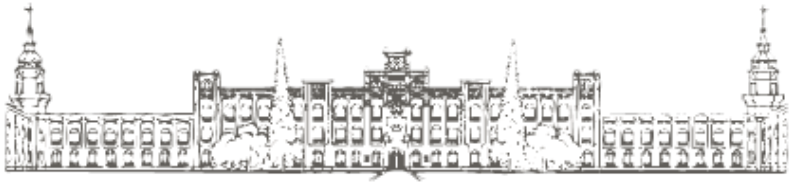

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Моделювання систем. Курсова робота»
Тема: Метод оптимізації параметрів імітаційної моделі еволюційним методом

Керівник:
Дифучин А.Ю.

«Допущено до захисту»
_____

«___» _____ 2023 р.

Захищено з оцінкою
_____

Члени комісії:

_____ Інна СТЕЦЕНКО

_____ Антон ДИФУЧИН

Виконавець:
Галько Міла Вячеславівна
студентка групи ІП-01
залікова книжка № 0107
_____

«12» грудня 2023 р.

Київ – 2023

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Кафедра інформатики та програмної інженерії

Дисципліна «Моделювання систем»

Спеціальність 121 Інженерія програмного забезпечення

Курс 4 Група ІП-01 Семестр 1

ЗАВДАННЯ

.

на курсову роботу студентки

Галько Міли Вячеславівни

(прізвище, ім’я, по батькові)

1. Тема роботи «Метод оптимізації параметрів імітаційної моделі еволюційним методом

2. Термін здачі студентом закінченої роботи "12" грудня_ 2023р.

3. Зміст розрахунково-пояснювальної записки

1. Опис генетичного алгоритму 2. Псевдокод генетичного алгоритму 3. Реалізація генетичного алгоритму 4. Реалізація Model1 5. Реалізація Model2 6. Проведення експериментів над моделями. Висновки.

4. Дата видачі завдання "6" жовтня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів виконання курсової роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	06.10.2023	
2	Формулювання теми курсової роботи	09.10.2023	
3	Аналіз принципу роботи генетичного алгоритму	16.10.2023	
4	Програмна розробка генетичного алгоритму для оптимізації параметрів імітаційної моделі	25.10.2023	
5	Розробка моделей	06.11.2023	
6	Виконання експериментів над моделями	15.11.2023	
8	Оформлення пояснювальної записки	30.11.2023	
9	Подання КР на перевірку	08.12.2023	
10	Захист КР	12.12.2023	

Студент _____ Галько Міла

(підпис)

Керівник _____ Дифучин А.Ю.

(підпис)

АНОТАЦІЯ

Курсова робота: 78 с., 12 рис., 6 табл., 2 додатки, 4 джерела літератури.

Мета роботи: розробка та оптимізація параметрів імітаційної моделі шляхом використання еволюційного методу. Основні завдання включають вивчення функціонування системи та створення формалізованої моделі для подальших досліджень, а також аналіз впливу різних параметрів на поведінку системи.

У розділі 1 "Опис генетичного алгоритму" розглядаються основні принципи та концепції генетичного алгоритму, який використовується для оптимізації параметрів імітаційної моделі.

В розділі 2 "Псевдокод генетичного алгоритму" подано деталізований псевдокод алгоритму, який визначає кроки та процеси оптимізації параметрів за допомогою еволюційного методу.

Розділ 3 "Реалізація генетичного алгоритму" надає опис програмної реалізації генетичного алгоритму, включаючи вибір функції пристосованості, мутації та кросоверу.

Розділи 4 та 5 "Реалізація Model1" і "Реалізація Model2" детально описують структуру та особливості реалізації об'єктів систем масового обслуговування, які є об'єктами оптимізації у рамках генетичного алгоритму.

У розділі 6 "Проведення експериментів над моделями" подано результати експериментів, в яких вивчається вплив різних параметрів на ефективність системи, а також порівняння роботи обох моделей.

КЛЮЧОВІ СЛОВА: ГЕНЕТИЧНИЙ АЛГОРИТМ, ОПТИМІЗАЦІЯ ПАРАМЕТРІВ, ІМІТАЦІЙНА МОДЕЛЬ, СИСТЕМА МАСОВОГО ОБСЛУГОВУВАННЯ, ЕВОЛЮЦІЙНИЙ МЕТОД, ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

ЗМІСТ

ВСТУП

Курсова робота спрямована на дослідження та використання генетичного алгоритму для оптимізації параметрів імітаційної моделі систем масового обслуговування (СМО). Головною метою є створення універсального алгоритму, який ефективно адаптується до різноманітних умов та різних типів об'єктів обслуговування [1].

У даному дослідженні генетичний алгоритм використовується для еволюційного підбору оптимальних параметрів імітаційної моделі. Це враховує різноманітність умов та індивідуальні особливості об'єктів обслуговування, що дозволяє досягти покращення ефективності та оптимальності системи [3].

Робота передбачає розробку алгоритму мовою програмування C# та створення необхідної інфраструктури для вимірювання та аналізу статистики роботи СМО під час еволюційного процесу оптимізації параметрів. Після реалізації алгоритму буде проведено дослідження його роботи на прикладах конкретних моделей масового обслуговування.

Отримані результати не лише доведуть коректність роботи системи, але й дозволяють визначити оптимальні параметри (визначаються користувачем), що оптимізують процес обслуговування в конкретних умовах та вимогах. Це відкриє перспективи для подальшого вдосконалення та розширення функціонала розробленого генетичного алгоритму в контексті моделювання систем масового обслуговування.

1 ОПИС ГЕНЕТИЧНОГО АЛГОРИТМУ

Генетичний алгоритм є потужним інструментом для розв'язання задач оптимізації параметрів імітаційних моделей. В цьому розділі буде надано детальний опис використаного генетичного алгоритму для оптимізації параметрів імітаційної моделі системи масового обслуговування.

1.1 Основні принципи генетичного алгоритму

Генетичний алгоритм базується на принципах природного відбору та еволюції, використовуючи поняття генетичних операцій, таких як селекція, кросовер та мутація. В рамках нашої роботи генетичний алгоритм використовується для ефективного підбору оптимальних параметрів імітаційної моделі системи масового обслуговування. [2]

1.2 Структура генетичного алгоритму

Генетичний алгоритм включає такі основні етапи:

1. Ініціалізація популяції – це створення початкової популяції імітаційних моделей з різними наборами параметрів;
2. Оцінювання пристосованості – процес надання кожній імітаційній моделі рейтингу відповідно до її пристосованості, яка визначається за певною функцією оцінювання;
3. Селекція – процес формування нового покоління з попередніх моделей на основі їхньої пристосованості. Вибір відбувається таким чином, що більш пристосовані моделі мають більше шансів потрапити у нове покоління;
4. Кросовер – процес спрямований на об'єднання параметрів обраних випадковим чином моделей для створення нових імітаційних моделей.
5. Мутація – піддавання мутаціям моделей, що включає випадкові зміни параметрів для розширення різноманітності.

1.3 Висновки

Для ефективної роботи генетичного алгоритму важливо визначити параметри, такі як розмір популяції, ймовірності кросовера та мутації, що відповідають особливостям досліджуваної системи, та кількість ітерацій генетичного алгоритму.

2 ПСЕВДОКОД ГЕНЕТИЧНОГО АЛГОРИТМУ

У цьому розділі розглянемо детальний опис роботи еволюційного алгоритму, який використовується для оптимізації параметрів імітаційних моделей систем масового обслуговування.

Оскільки еволюційний алгоритм складається з конкретних етапів, які забезпечують ефективний пошук оптимальних параметрів імітаційної моделі, то псевдокод буде розділений на відповідні частини та продемонстрований в черзі їх виконуваності в алгоритмі.

2.1 Ініціалізація популяції

На цьому етапі створюється вихідна популяція імітаційних моделей із випадковими параметрами:

```
function InitializePopulation():
    population = []
    for i in range(PopulationSize):
        individual = createRandomIndividual()
        population.append(individual)
    return population
```

2.2 Оцінювання пристосованості

Кожній імітаційній моделі надається оцінка пристосованості (fitness) на основі її відповіді під час симуляції:

```
function EvaluatePopulation(population):
    for individual in population:
        individual.fitness = simulateAndEvaluate(individual)
```

2.3 Селекція

Селекція вибирає батьків для наступного покоління на основі їхньої пристосованості:

```
function TournamentSelection(population):
    selectedParents = []
    for i in range(PopulationSize):
        contestant1, contestant2 = randomSelection(population)
        selectedParent = selectBetter(contestant1, contestant2)
        selectedParents.append(selectedParent)
    return selectedParents
```

2.4 Кросовер

На цьому етапі проводиться кросовер (обмін генами) між обраними батьками для створення нового покоління:

```
function crossover(parents):
    offspring = []
    for i in range(0, PopulationSize, 2):
        parent1, parent2 = Parents[i], Parents[i + 1]
        if random() < CrossoverProbability:
            child1, child2 = performCrossover(parent1, parent2)
            offspring.append(child1)
            offspring.append(child2)
        else:
            offspring.append(parent1)
            offspring.append(parent2)
    return offspring
```

2.5 Мутація

Мутація випадковим чином змінює окремі параметри імітаційних моделей для збереження різноманітності в популяції [4]:

```
function Mutate(offspring):
```

```
for individual in offspring:
    if random() < MutationProbability:
        performMutation(individual)
```

3 РЕАЛІЗАЦІЯ ГЕНЕТИЧНОГО МЕТОДУ

3.1 Застосування моделей

З метою використання генетичного методу для оптимізації параметрів імітаційних моделей систем масового обслуговування, в роботі впроваджено інтерфейс IGeneticModel. Цей інтерфейс служить основою для реалізації конкретних моделей, які піддаватимуться оптимізації.

Інтерфейс містить методи, необхідні для взаємодії з генетичним алгоритмом:

- Mutate(): реалізує зміни в параметрах моделі для проведення мутацій;
- Simulate(): запускає симуляцію роботи моделі;
- CalculateProfit(): обчислює показник прибутковості моделі, який буде використовуватися для оцінки її ефективності в генетичному алгоритмі.

Проте сам клас GeneticAlgorithm, що реалізує алгоритм, буде використовувати об'єкти типу Individual<T>, де T є конкретною реалізацією інтерфейсу IGeneticModel. Об'єкти цього класу представляють індивідууми в популяції, які піддаються еволюції для знаходження оптимальних параметрів систем масового обслуговування. Сам клас містить:

- T GeneticModel: Представляє конкретну реалізацію інтерфейсу IGeneticModel, яка визначає структуру та параметри імітаційної моделі.
- Fitness: значення придатності індивідуума, яке визначається показником прибутковості моделі. Це значення використовується для оцінки ефективності індивідуума в рамках генетичного алгоритму.

3.2 Реалізація кросовера

Для ефективного застосування генетичного алгоритму та забезпечення різноманітності нових індивідуумів в популяції впроваджено інтерфейс ICrossoverLogic. Цей інтерфейс визначає метод Crossover, який буде відповідати

за виконання кросовера (схрещування) між двома батьківськими індивідуумами для створення нащадку.

Логіка кросовера винесена в окремий інтерфейс ICrossoverLogic з метою забезпечення гнучкості та можливості легкої заміни конкретної реалізації кросовера без значних змін у загальній структурі генетичного алгоритму. Це дозволяє досліджувати та порівнювати різні методи кросовера для досягнення оптимальних результатів у конкретному контексті.

Окрема логіка кросовера може бути реалізована шляхом створення класів, які реалізують інтерфейс ICrossoverLogic<T>. Кожен такий клас визначає свій унікальний метод кросовера відповідно до специфіки області застосування.

Такий підхід забезпечує модульність системи та дозволяє з легкістю впроваджувати нові алгоритми кросовера чи оптимізувати чинний без великих змін у загальній структурі генетичного алгоритму.

3.3 Застосування фабрики

Інтерфейс IGeneticModelFactory<T> виконує ключову роль у структурі генетичного алгоритму для оптимізації параметрів імітаційної моделі. Його завданням є створення початкових індивідуумів (моделей) для подальшої оптимізації генетичним алгоритмом.

Такий підхід дозволяє використовувати різні фабрики для різних типів імітаційних моделей. Кожна фабрика може мати свою унікальну логіку генерації вихідних параметрів, що впливає на манеру дослідження та оптимізації конкретної моделі.

Виокремлення створення індивідуумів у фабрики дозволяє забезпечити гнучкість та легкість розширення системи. Заміна фабрики для конкретної моделі може бути здійснена без внесення змін у загальну логіку генетичного алгоритму.

4 РЕАЛІЗАЦІЯ MODEL1

4.1 Опис Model1 та її параметрів

Створимо Модель Model1, що буде є простою системою масового обслуговування для наочності правильності роботи генетичного алгоритму. Вона включає в себе етапи створення об'єктів, їх обробку у певному порядку та вивільнення ресурсів. Схематично модель зображена на рисунку 4.1.

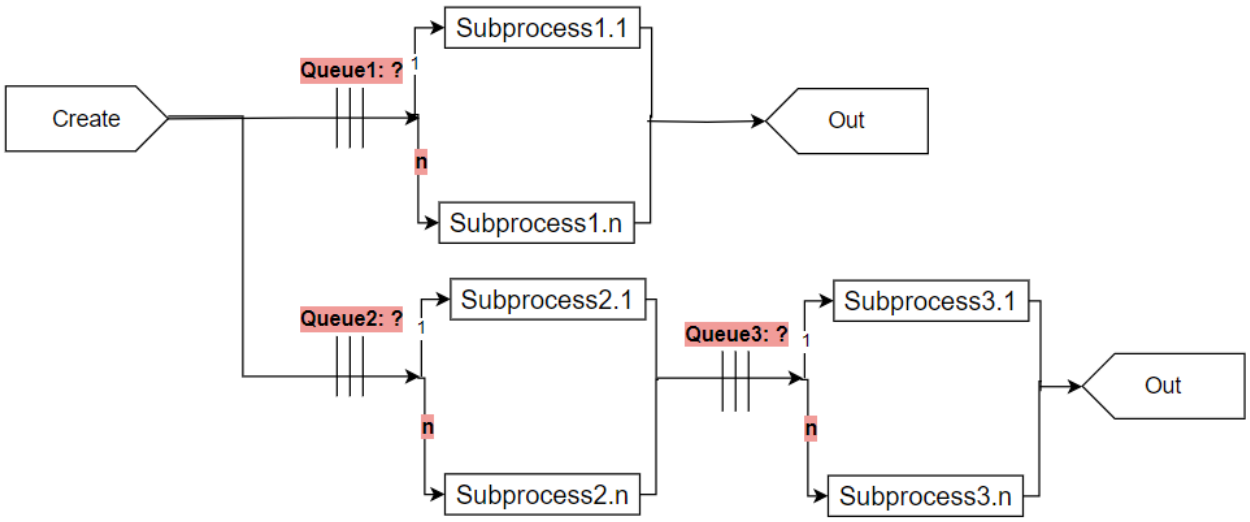

Рисунок 4.1 – Схема Model1 із позначенням параметрів для генетичного алгоритму

З рисунка видно, що основними елементами виступають 1 елемент, що створює об’єкти, та 3 процеси, що оброблюють. Кожен процес може містити чергу об’єктів та визначену кількість підпроцесів. Саме ці параметри й будуть досліджені у генетичному алгоритмі.

4.2 Програмна реалізація Model1

4.2.1 Параметри Model1 та її структура

Як було зазначено в розділі 3.1 клас Model1 буде реалізовувати інтерфейс IGeneticModel. Вона буде утримувати імена параметрів в наочному вигляді:

```
public enum Names {
```

```
            SubProcessCount1, MaxQueue1,
            SubProcessCount2, MaxQueue2,
            SubProcessCount3, MaxQueue3
        }
```

Самі ж параметри будуть визначені як Dictionary<Names, int> Data.

Для створення моделі створимо метод CreateModel:

```
    public static Model CreateModel(Dictionary<Names, int> data) {
        Create create = new(delay: 1);
        Process process1 = new(50, data[Names.SubProcessCount1], "Process1",
data[Names.MaxQueue1]);
        Process process2 = new(7, data[Names.SubProcessCount2], "Process2",
data[Names.MaxQueue2]);
        Process process3 = new(12, data[Names.SubProcessCount3], "Process3",
data[Names.MaxQueue3]);
        var container = new NextElementsContainerByQueuePriority();
        container.AddNextElement(process1, 1);
        container.AddNextElement(process2, 2);
        create.NextElementsContainer = container;
        process2.NextElementsContainer = new NextElementContainer(process3);
        return new Model(new List<Element> { create, process1, process2, process3 });
    }
```

4.2.2 Генерація параметрів Model1

У розділі 3.3 був зазначений інтерфейс IGeneticModelFactory, який буде реалізований у вигляді Model1Factory для генерації Model1. Загалом для кожного параметра черги будемо у відповідність ставити число від 0 до 30 включно, а для підпроцесів від 1 до 30.

4.2.3 Мутація параметрів Model1

Мутація буде змінювати випадковий параметр на ціле число від -1 до 1. Умова при цьому така, що якщо нові параметри не можуть існувати імітація повторюється для початкового значення параметрів:

```
Procedure Mutate():
    while True
        parameterIndex = Random.Shared.Next(Data.Count)
        adder = Random.Shared.Next(-1, 1)
        Data[Enum.GetValues<Names>()[parameterIndex]] += adder
        if DataIsValid(Data) then Exit Loop
        Data[Enum.GetValues<Names>()[parameterIndex]] -= adder
    _model = CreateModel(Data)
End Procedure
```

4.2.4 Кросовер Model1

Для реалізації етапу схрещування створимо реалізацію інтерфейсу ICrossoverLogic<T>, де T – Model, у вигляді класу Model1Crossover.

Саме схрещування відбувається шляхом взяття першої половини параметрів від parent1, а другої – від parent2:

```
Function Crossover(parent1: Model1, parent2: Model1): Model1
    childData = Dictionary<Model1.Names, int>()
    for i = 0 to parent1.Data.Count - 1
        if i < parent1.Data.Count / 2
            childData[Enum.GetValues<Model1.Names>()[i]] =
parent1.Data[Enum.GetValues<Model1.Names>()[i]]
        else childData[Enum.GetValues<Model1.Names>()[i]] =
parent2.Data[Enum.GetValues<Model1.Names>()[i]]
    return new Model1(childData, parent1.SimulationTime)
End Function
```

4.3 Визначення границь та значущості параметрів Model1

Валідність значень:

- черга більш як 0 включно;
- кількість підпроцесів більш як 1 включно.

Профіт (значущість гена) вираховується як сума добутків кожного з параметрів та параметра процента відмови моделі на визначені коефіцієнти.

5 РЕАЛІЗАЦІЯ MODEL2 (лікарня)

Model2 відображає вже складнішу задачу, а саме імітацію роботи лікарні із різним типом об’єктів «Хворий» та їх обробкою відповідно.

5.1 Опис Model2 та її параметрів

У лікарню поступають хворі таких трьох типів:

1) хворі, що пройшли попереднє обстеження і направлені на лікування – тип Chamber;

2) хворі, що бажають потрапити в лікарню, але не пройшли повністю попереднє обстеження – тип NotExamined;

3) хворі, які тільки що поступили на попереднє обстеження – тип Lab.

Таблиця 5.1 – Чисельні характеристики типів хворих

Тип хворого	Відносна частота	Середній час реєстрації, хв
1	0,5	15
2	0,1	40
3	0,4	30

При надходженні в приймальне відділення хворий стає в чергу, якщо обидва чергових лікарі зайняті. Лікар, який звільнився, вибирає в першу чергу хворого типу 1.

Після заповнення різноманітних форм у приймальне відділення хворі 1 типу ідуть прямо в палату, а хворі типів 2 і 3 направляються в лабораторію. Троє супровідних розводять хворих по палатах. Хворим не дозволяється направлятися в палату без супровідного. Якщо всі супровідні зайняті, хворі очікують їхнього звільнення в приймальному відділенні. Як тільки хворий доставлений у палату, він вважається таким, що завершив процес приймання у до лікарні. Хворі, що спрямовуються в лабораторію, не потребують супроводу. Після прибуття в лабораторію хворі стають у чергу в реєстратуру. Після реєстрації вони ідуть у

кімнату очікування, де чекають виклику до одного з двох лаборантів. Після здачі аналізів хворі або повертаються в приймальне відділення (хворий типу 2), або залишають лікарню (хворий типу 1). Після повернення в приймальне відділення хворий, що здав аналізи, розглядається як хворий типу 1.

Таблиця 5.2 – Дані по тривалості дій (хв)

Величина	Розподіл
Час між прибуттями в приймальне відділення	Експоненціальний з математичним сподіванням 15
Час слідування в палату	Рівномірне від 3 до 8
Час слідування з приймального відділення в лабораторію і навпаки	Рівномірне від 2 до 5
Час обслуговування в реєстратуру лабораторії	Ерланга з математичним сподіванням 4,5 і k=3
Час проведення аналізу в лабораторії	Ерланга з математичним сподіванням 4 і k=2

Визначити час, проведений хворим у системі, тобто інтервал часу, починаючи з надходження і закінчуючи доставлянням в палату (для хворих типу 1 і 2) або виходом із лабораторії (для хворих типу 3). Визначити також інтервал між прибуттями хворих у лабораторію.

Отже, отримуємо схематично роботу лікарні на рисунку 5.1.

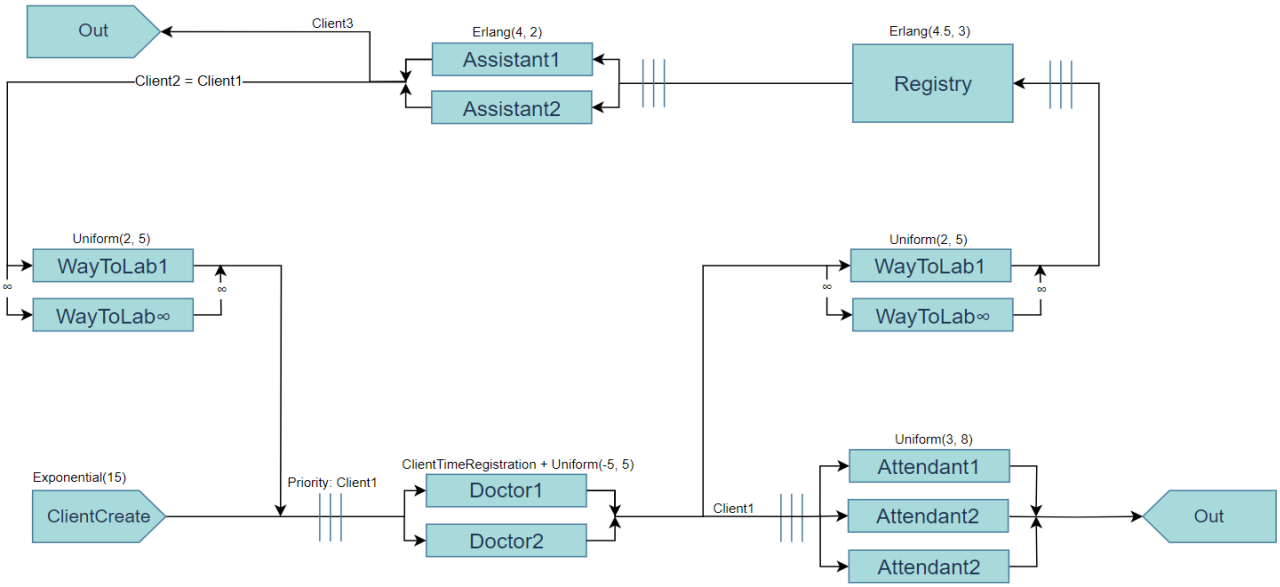

Рисунок 5.1 – Схема Model2

Для покращення роботи лікарні необхідно виявити параметри, які можемо покращувати користуючись алгоритмом. Оскільки установа має працівників, саме їх кількість і буде визначена для зменшення черг. Отже, будемо зосереджуватись на:

- виявленні мінімальної та при цьому оптимальної кількості наступних типів робітників: лікар, супровідний, працівник реєстратури, асистент лабораторії;
- зменшенні людей у чергах перед обслуговуванням працівником лікарні;
- неможливості відмови обслуговування (FailurePercentage).

Припустимо, що час роботи кожного спеціаліста є визначеним і не може змінюватись. Так само ми не контролюємо наскільки швидко клієнт лікарні витрачає часу на перехід від лікарні до лабораторії та навпаки. Отже, замовник, що потребує поліпшення роботи системи, хоче зменшити час обстеження хворих, при цьому для його бюджету буде краща менша кількість працівників.

5. 2 Реалізація особливостей поведінки Model2

Оскільки модель Model2 відрізняється від класичної, і вимагає деяких змін у поведінці її елементів, розпочнімо з визначення пацієнтів (хворих) у системі.

5.2.1 Визначення пацієнтів

У моделі Model2 пацієнти представлені об'єктом `Client` (наслідується від Item, що обробляється в класичній моделі), який утримує час його обробки лікарем та тип хворого з наступного переліку:

```
public enum ClientType {
    Chamber,
    NotExamined,
    Lab
}
```

Саме ці об’єкти створює спадкоємець CreateClient від звичайного Create:

```
public class CreateClient : Create {
    public CreateClient(Randomizer randomizer, string name) :
base(randomizer, name) {}
    protected override Item CreateItem() {
        var r = new Random().NextDouble();
        if (r <= 0.5) return new Client("ChamberClient", ClientType.Chamber,
15, Time.Curr);
        if (r <= 0.6) return new Client("NotExaminedChamberClient",
ClientType.NotExamined, 40, Time.Curr);
        return new Client("LabClient", ClientType.Lab, 30, Time.Curr);
    }
}
```

5.2.2 Реалізація процесу DoctorProcess

Оскільки класичний елемент «Процес» обирає з поля ItemsQueue об’єкти по черзі, то маємо змінити цю поведінку для DoctorProcess. Створимо нащадка ClientsQueue від ItemsQueue, який дозволить в першу чергу обирати хворих типу Chamber. Об’єкт наступного класу і буде визначеним у класі DoctorProcess замість звичайної черги:

```
internal class ClientsQueue : ItemsQueue {
    public ClientsQueue(int limit) : base(limit) {}
    public override Item GetItem() {
        var chamberClient = Queue.Find(x => (x as Client).ClientType ==
ClientType.Chamber);
        if (chamberClient != null) {
            Queue.Remove(chamberClient);
            return chamberClient;
        }
        return base.GetItem();
    }
}
```

Подібним чином змінимо клас NextElementContainer, що утримував посилання на наступний елемент в системі. Оскільки в залежності від типу хворого NextElementContainer буде мати різні, проте визначені посилання:

```
public class NextAfterDoctor : NextElementsContainer {
    private Process NextAttendantProcess { get; }
    private Process NextWayToLab { get; }
    private DoctorProcess _doctorProcess;
    public NextAfterDoctor(DoctorProcess doctorProcess, Process
nextAttendantProcess, Process nextWayToLab) {
        _doctorProcess = doctorProcess;
        NextAttendantProcess = nextAttendantProcess;
        NextWayToLab = nextWayToLab;
    }
    protected override Element GetNextElement() {
        var client = _doctorProcess.Item as Client;
        return client.ClientType == ClientType.Chamber ? NextAttendantProcess
: NextWayToLab;
```

```
    }
}
```

5.2.3 Реалізація процесу LabAssistanceProcess

Клас LabAssistanceProcess також наслідується від звичайного Process, проте необхідно перевизначити його поведінку перед завершенням обслуговування клієнта. Змінимо тип клієнта NotEximined на Chamber, а для LabClient наступним елементом системи буде вихід з неї:

```
public class LabAssistanceProcess : Process {
    public LabAssistanceProcess(Randomizer randomizer, int assistanceCount, string name, string subProcessName, int maxQueue = 2147483647) : base(randomizer, assistanceCount, name, maxQueue, subProcessName) {}

    protected override void NextElementsContainerSetup() {
        if (Item is Client { ClientType: ClientType.NotExamined } client) {
            client.ClientType = ClientType.Chamber;
            client.RegistrationTime = 15;
            client.Name = "ChamberClient";
        }
        else NextElementsContainer = null;
    }
}
```

5.3 Програмна реалізація Model2

5.3.1 Параметри Model2 та її структура

Подібним чином до Model1 визначимо й Model2 через інтерфейс IGeneticModel із наступним переліком імен параметрів генетичного алгоритму:

```
public enum Names {
```

```
            DoctorsCount, DoctorsQueue,
            AttendantsCount, AttendantsQueue,
            RegistryWorkersCount, RegistryQueue,
            AssistantsCount, AssistantsQueue,
        }
```

Параметри утримуються в полі Dictionary<Names, int> Data і застосовуються для створенні моделі:

```
public static Model CreateModel(Dictionary<Names, int> data)
   {
      CreateClient patients = new(new ExponentialRandomizer(15), "Patient");
      DoctorProcess doctors = new(doctorsCount: data[Names.DoctorsCount],
"Doctors", "Doctor", data[Names.DoctorsQueue]);
      Process attendants = new(new UniformRandomizer(3, 8),
data[Names.AttendantsCount], "Attendants", data[Names.AttendantsQueue],
"Attendant");
      Process fromHospitalToLab = new(new UniformRandomizer(2, 5), 25, name:
"WayToLab");
      Process labRegistry = new(new ErlangRandomizer(4.5, 3),
data[Names.RegistryWorkersCount], "Registry", data[Names.RegistryQueue],
"RegistryWorker");
      LabAssistanceProcess labAssistants = new(new ErlangRandomizer(4, 2),
data[Names.AssistantsCount], "Assistants", "Assistant",
data[Names.AssistantsQueue]);
      Process fromLabToHospital = new(new UniformRandomizer(2, 5), 25, name:
"WayToHospital");

      patients.NextElementsContainer = new NextElementContainer(doctors);
      doctors.NextElementsContainer = new NextAfterDoctor(doctors, attendants,
fromHospitalToLab);
```

```
        fromHospitalToLab.NextElementsContainer = new
NextElementContainer(labRegistry);
        labRegistry.NextElementsContainer = new
NextElementContainer(labAssistants);
        labAssistants.NextElementsContainer = new
NextElementContainer(fromLabToHospital);
        fromLabToHospital.NextElementsContainer = new
NextElementContainer(doctors);

        return new Model(new List<Element>(){ patients, doctors, attendants,
fromHospitalToLab, labRegistry, labAssistants, fromLabToHospital });
}
```

5.3.2 Генерація параметрів Model2

Реалізуємо фабрику Model2Factory, що буде працювати за наступним алгоритмом:

```
Method createRandomModel():
   data = new Dictionary()
   data[Model2.Names.DoctorsCount] = randomInteger(1, 5)
   data[Model2.Names.DoctorsQueue] = randomInteger(0, 100)
   data[Model2.Names.AttendantsCount] = randomInteger(1, 5)
   data[Model2.Names.AttendantsQueue] = randomInteger(0, 100)
   data[Model2.Names.RegistryWorkersCount] = randomInteger(1, 5)
   data[Model2.Names.RegistryQueue] = randomInteger(0, 100)
   data[Model2.Names.AssistantsCount] = randomInteger(1, 5)
   data[Model2.Names.AssistantsQueue] = randomInteger(0, 100)
   return new Model2(data, 1000)
```

Отже, як видно з генерації, люба черга може приймати значення від 0 до 100, а кількість робітників на одну посаду – від 1 до 5.

5.3.3 Мутація параметрів Model2

Функція мутації ідентична до методу Mutate класу Model1 (4.2.3). Мутація відбувається за рахунок зміни випадкового параметра на число з діапазону від -1 до 1. Якщо нові дані невалідні, повторюється процес мутації над початковими даними.

5.3.4 Кросовер Model2

Схема схрещування параметрів моделі відбувається подібним чином до Model1 у Model1Crossover (розділ 4.2.4). Береться перша половина параметрів від parent1 та друга – від parent2.

5.4 Визначення границь та значущості параметрів Model2

Валідність значень:

1) черга більше 0 включно;
2) кількість підпроцесів більше 1 включно.

Профіт (значущість гена) вираховується як сума добутків кожного з параметрів Data та параметра процента відмови моделі на визначені коефіцієнти.

6 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТІВ НАД МОДЕЛЯМИ

Головною задачею цього розділу буде демонстрація роботи генетичного алгоритму при зміні коефіцієнтів при обчисленні значущості гена (профіт). Маємо наступні параметри генетичного алгоритму:

1) PopulationSize = 400;
2) MaxGenerations = 100;
3) CrossoverProbability = 0.7;
4) MutationProbability = 0.1.

6.1 Експеримент над Model1

Для початку визначимо, яких результатів планується досягнути при оптимізації моделі в переліку пріоритетності:

1) зменшення показника відмови;
2) мінімізація черги;
3) зменшення кількості підпроцесів.

Отже, базуючись на наших вищеописаних критеріях необхідно підібрати значення коефіцієнтів для методу CalculateProfit. Сама функція підрахунку має наступний вигляд:

$$\begin{aligned} \text{Result} = \quad & \text{SubProcessCount1} * \text{k1} + \text{MaxQueue1} * \text{k2} \\ & + \text{SubProcessCount2} * \text{k3} + \text{MaxQueue2} * \text{k4} \\ & + \text{SubProcessCount3} * \text{k5} + \text{MaxQueue3} * \text{k6} \\ & + \text{FailurePercent} * \text{k7} \end{aligned} \tag{6.1}$$

Очевидно, що усі коефіцієнти мають бути від’ємними, оскільки через задачу мінімізації усіх параметрів необхідно використовувати спадну функцію. Бо, якщо цього не зробити, алгоритм просто збільшить усі ресурси до максимально можливого значення, щоб отримати максимальний профіт. Однак для забезпечення балансу та уникнення такого перевантаження системи, кожний коефіцієнт повинен бути обернено пропорційний відповідному параметру.

Нехай спробуємо для усіх коефіцієнтів обрати базове значення, таке як -1. При таких коефіцієнтах отримуємо наступний вивід на рисунку 6.1:

```
Generation 0: Best Fitness = -55
Generation 1: Best Fitness = -51.616784630940344
Generation 2: Best Fitness = -46.59820538384845
Generation 3: Best Fitness = -39.21459227467811
Generation 4: Best Fitness = -39.21436227224009
Generation 5: Best Fitness = -35.65726227795193
Generation 6: Best Fitness = -38.75253549695741
Generation 7: Best Fitness = -36.71715145436309
Generation 8: Best Fitness = -34.12992545260916
Generation 9: Best Fitness = -36.17622950819672
Generation 10: Best Fitness = -34.53968253968254
Generation 11: Best Fitness = -34.53968253968254
Generation 12: Best Fitness = -34.005181347150256
Generation 13: Best Fitness = -33.45398773006135
Generation 14: Best Fitness = -32.956043956043956
Generation 15: Best Fitness = -33.644158628081456
Generation 16: Best Fitness = -33.644158628081456
Generation 17: Best Fitness = -33.75229357798165
Generation 18: Best Fitness = -33.71739130434783
Generation 19: Best Fitness = -33.71444082519001
Generation 20: Best Fitness = -32.59914712153518
Generation 21: Best Fitness = -31.588983050847457
Generation 22: Best Fitness = -31.588983050847457
Generation 23: Best Fitness = -33.21518987341772
Generation 24: Best Fitness = -33.21285563751317
Generation 25: Best Fitness = -33.21052631578947
Generation 43: Best Fitness = -33.42476970317298
Generation 44: Best Fitness = -32.90707497360084
Generation 45: Best Fitness = -32.28389830508475
Generation 46: Best Fitness = -32.28042328042328
Generation 47: Best Fitness = -32.276955602537
```

Рисунок 6.1 – Вивід найкращого профіта для перших 47 генерацій Model1

Видно, що результат значущості гена йде в сторону поліпшення. Так значення стартувало з -55 і на 47 ітерації покращується до -32.28. Звичайно, через етап відбору батьків для наступної генерації, який по суті є випадковою вибіркою серед попереднього етапу, отримуємо іноді погіршення значення найкращого профіту покоління.

Нагадаємо, що метод генетичного алгоритму TournamentSelection виконується шляхом обирання двох випадкових індивідуумів і серед них обирається 1 з кращим значенням значущості. Дане рішення обґрунтовано тим, що схрещування найкращих може не дати кращих результатів. В свою чергу,

найгірші можуть дати гарного потомка. Подібна випадковість не заганяє генерацію до тупика і дає можливість знайти неочікувані результати.

Самі ж результати демонструють найкращого індивідуума за весь час роботи алгоритму на рисунку 6.2:

```
------------RESULTS------------
CREATE:
        Quantity = 970
Process1:
        WorkTime = 1
        InActQuantity = 74
        OutActQuantity = 22
        Current queue length = 1
        Mean length of queue = 0.9844434358567702
        Failure probability = 0.6756756756756757
        SubProcess_0:
                Quantity = 23
                WorkTime = 1.00000
Process2:
        WorkTime = 0.9979206595660849
        InActQuantity = 896
        OutActQuantity = 887
        Current queue length = 0
        Mean length of queue = 0.053702716599991374
        Failure probability = 0
        SubProcesses Total: count=10 quantity=896
Process3:
        WorkTime = 0.994765769667913
        InActQuantity = 887
        OutActQuantity = 865
        Current queue length = 0
        Mean length of queue = 0.01608329697689126
        Failure probability = 0.018038331454340473
        SubProcesses Total: count=16 quantity=871

SubProcessCount1: 1
 MaxQueue1: 1
SubProcessCount2: 10
 MaxQueue2: 1
SubProcessCount3: 16
 MaxQueue3: 1
FailurePercent: 1.520086862106406,
```

Рисунок 6.2 – Результати тестування №1 для Model1

Одразу видно, що значення тестованих параметрів не є максимальними. З опису моделі можемо зазначити, що усі процеси працюють в середньому 99% часу, отже алгоритм зміг досягнути збалансованого навантаження системи.

Для аналізу найкращої кількості підпроцесів, необхідно згадати затримки елементів моделі з таблиці 6.1.

Таблиця 6.1 – Час delay для елементів Model1

Елемент	Час затримки/обробки
Create	1
Process1	50
Process2	7
Process3	12

Оскільки Create працює дуже швидко, а Process1 та Process2 більш повільно, то очевидно, що це призведе до великої зміни кількості підпроцесів. В нашому випадку ця кількість сягає 1 для Process1 та 16 для Process2. Така різниця обґрунтована занадто довгою обробкою об’єктів Process1. Отже, алгоритм, вирішив що поки працює 1 довгий процес, будуть виконуватися 16 коротких. Через це об’єкти після Create не формують довгу чергу і з ймовірністю 98,5% одразу потрапляють на обробку далі.

Відповідно до поставленої задачі будемо зменшувати помилки та черги до нуля та кількість підпроцесів в пріоритеті для 2-го, 3-го та 1-го процесу. Отже, встановимо наступні коефіцієнти відображені на таблиці 6.2 та проведемо із ними декілька тестів (рисунок 6.3-6.4).

Таблиця 6.2 – Значення коефіцієнтів тестування №2 для Model1

Параметр	Значення коефіцієнта
FailurePercent	-1000
SubProcessCount2	-10
SubProcessCount3	-7
SubProcessCount1	-6
MaxQueue1	-100
MaxQueue2	-100
MaxQueue3	-100

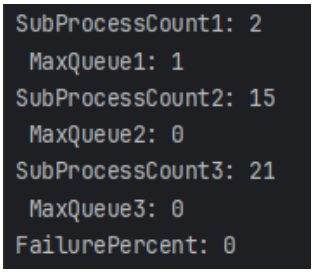

Рисунок 6.3 – Результати Model1 із вірогідністю схрещування 0.7 та мутації 0.1

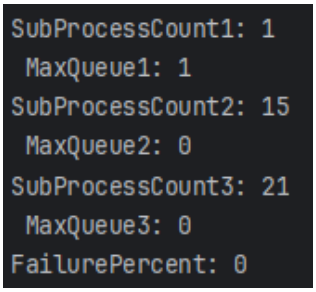

Рисунок 6.4 – Результати Model1 із вірогідністю схрещування 1.0 та мутації 0.5

Також, в якості експерименту, спробуємо аналог функції підрахунку значущості, результат якої зображено на рисунку 6.5, виду:

$$\begin{aligned}\text{Result} = {} & (\text{SubProcessCount1SubProcessCount2} + \text{SubProcessCount3}) \\ & * -1 + (\text{MaxQueue1} + \text{MaxQueue2} + \text{MaxQueue3}) * -100 \\ & + \text{FailurePercent} * -1000\end{aligned} \tag{6.2}$$

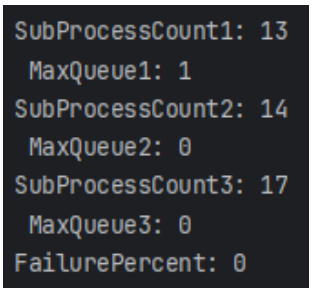

Рисунок 6.5 – Результати Model1 із вірогідністю схрещування 0.7 та мутації 0.1 з оновленою функцією CalculateProfit

6.2 Експеримент над Model2

В цьому розділі наведемо приклади оптимізацій моделі на основі двох запитів від замовника у вигляді задач.

6.2.1 Задача №1 «Пріоритет – клієнт»

Знову ж таки почнемо з визначення задачі, складової якої потребується вирішити в наступній пріоритетності:

1) зменшення вірогідності відмов;
2) зменшення часу пацієнта у системі;
3) зменшення кількості працівників.

Для розв’язання задачі сформуємо перелік параметрів та орієнтовні значення їх коефіцієнтів для підрахунку значущості індивідуума (таблиця 6.3).

Таблиця 6.3 – Значення коефіцієнтів для Model2 задача №1

Параметр	Значення коефіцієнта
DoctorsCount	-1
AttendantsCount	-1
RegistryWorkersCount	-1
AssistantsCount	-1
DoctorsQueue	-100
AttendantsQueue	-100
RegistryQueue	-100
AssistantsQueue	-100
AverageItemTimeInSystem	-1000
FailurePercent	-10 000

Тепер переглянемо результати алгоритму на рисунках 6.6-7.

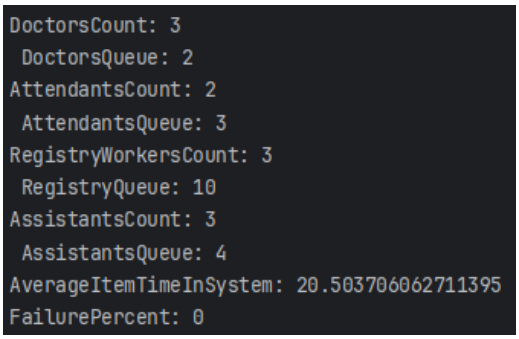

Рисунок 6.6 – Результати Model2 задача №1, 100 ітерацій та 400 популяція

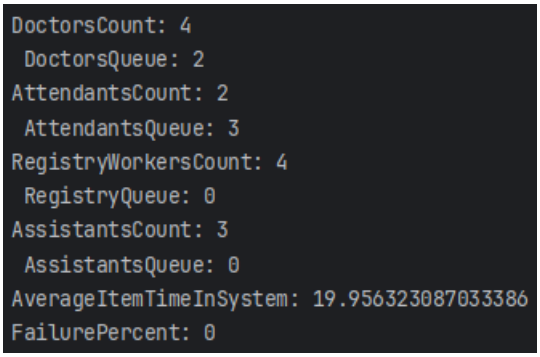

Рисунок 6.7 – Результати Model2 задача №1, 200 ітерацій та 700 популяція

Час хворого у системі сягає в середньому 20,5 хвилин на результатах рисунка 6.6. Проте існує досить велика черга в відділі лабораторії. В якості експерименту спробуємо збільшити коефіцієнт для RegistryQueue та AssistantsQueue до -200 (рисунок 6.8). І бачимо, що швидкість обслуговування в лабораторії зменшилась, а в лікарні навпаки. Відбувається більший прихід хворих до лікарні, що провокує збільшення кількості лікарів і черг у лікарні. Отже, цей варіант вже не підходить.

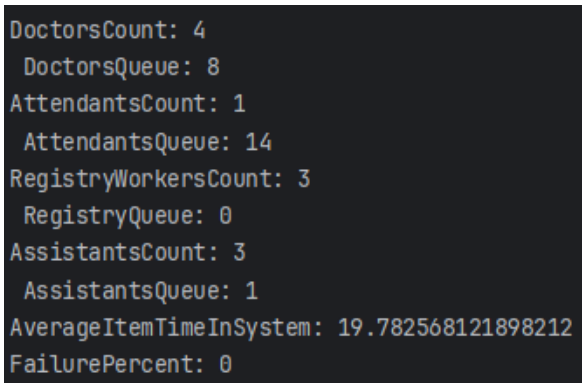

Рисунок 6.8 – Результати невдалого тесту №2 Model2 задача№1 із коефіцієнтом -200 для RegistryQueue та AssistantsQueue

Отримані результати по часу існування клієнта у системі не мають великих покращень. Дана задача слугує гарним прикладом того, як поліпшення параметрів одного сектору моделі може спричинити розбалансування її роботи в цілому.

6.2.2 Задача №2 «Пріоритет – гроші»

Нехай замовник потребує мінімізувати витрати. На основі цього сформуємо перелік аспектів, що потребують мінімізації в порядку важливості:

1) зменшення вірогідності відмов;
2) зменшення кількості працівників;
3) зменшення часу пацієнта у системі.

Спробуємо подібним чином до розв’язання задачі №1 визначити значення коефіцієнтів для поточного завдання (таблиця 6.4). Спробуємо виставити для черг коефіцієнти -1, проте для середнього часу хворого у системі визначимо більше значення, таке як -10. Адже, нам важливо мінімізація цього параметра, але шляхом кількості працівників чи створення черг – це рішення самої системи.

Таблиця 6.4 – Значення коефіцієнтів для Model2 задача №2

Параметр	Значення коефіцієнта
DoctorsCount	-100
AttendantsCount	-100
RegistryWorkersCount	-100
AssistantsCount	-100
DoctorsQueue	-1
AttendantsQueue	-1
RegistryQueue	-1
AssistantsQueue	-1
AverageItemTimeInSystem	-10
FailurePercent	-10 000

Тепер переглянемо результати алгоритму на рисунках 6.9-6.10, де кількість популяції була 700, а ітерацій – 200.

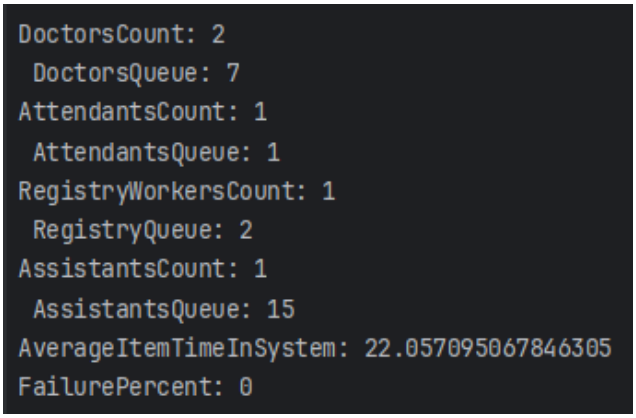

Рисунок 6.9 – Результати 1-го тесту Model2 задача №2

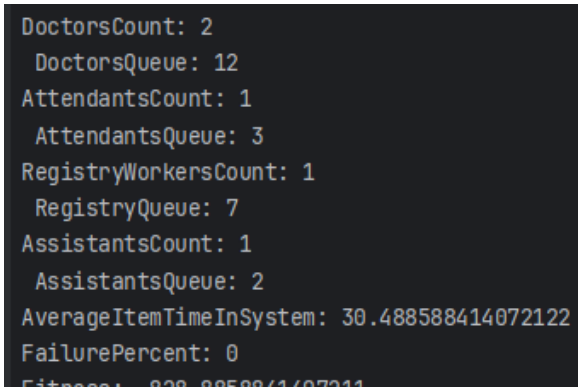

Рисунок 6.10 – Результати 2-го тесту Model2 задача №2

Загалом доволі різні результати часу клієнта у системі з однаковою кількістю робітників, може свідчити про те, що у день з рисунка 6.10 прийшло більше хворих типу 2. Це видно також і з залежності черг лікарів і у лабораторній у двох випадках.

Важливо зазначити, що кількість робітників однакова в обох випадках. Можливо це дійсно найбільш мінімізований персонал якого можна досягнути. На жаль, в залежності від того, якого типу буде приходити більше хворих певного типу, може утворюватися більше або менше черг.

ВИСНОВКИ

У ході виконання курсової роботи був досліджений метод оптимізації параметрів імітаційної моделі за допомогою генетичного алгоритму.

В першу чергу було розглянуто основні принципи роботи генетичного алгоритму, його етапи та параметри, що використовуються для його налаштування. Після чого було представлено конкретну реалізацію генетичного алгоритму для оптимізації параметрів імітаційних моделей. Також були зазначені інтерфейси, які використовуються для взаємодії з генетичним алгоритмом, такі як IGeneticModel, ICrossoverLogic, IGeneticModelFactory і т.д.

Для проведення тестувань було створено 2 моделі. Model1, що є простою системою масового обслуговування, була розроблена для демонстрації працездатності генетичного алгоритму. Також було розглянуто створення, параметризація та програмна реалізація даної моделі.

Далі була описана друга модель - Model2, яка відображає роботу лікарні з трьома типами пацієнтів. Розглядались особливості реалізації та модифікації, враховуючи різницю у структурі та логіці роботи базових елементів системи масового обслуговування із Model2.

Обидві моделі були внесені до генетичного алгоритму, де визначили параметри з метою максимізації значущості системи та забезпечення валідних значень параметрів. Також, були проведені додаткові ітерації коригування значень коефіцієнтів при обчисленні значущості гена на продемонстровані результати оптимізації. Отже, для кожної моделі визначили пріоритети оптимізації параметрів та були проведені серії тестів для знаходження оптимальних значень коефіцієнтів.

В результаті виконання курсової роботи було створення програмне забезпечення для автоматизованої оптимізації параметрів систем масового обслуговування за допомогою генетичного алгоритму. Важливо зазначити, що враховувати баланс між параметрами та вибирати їх значення з урахуванням специфіки конкретної задачі може призвести до покращення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Стеценко І.В. Навчальний посібник «Моделювання систем» [Електронний ресурс]. – Режим доступу: https://do.ipo.kpi.ua/pluginfile.php/112577/mod_resource/content/1/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8E%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%9D%D0%B0%D0%B2%D1%87%D0%9F%D0%BE%D1%81%D1%96%D0%B1%D0%BD%D0%B8%D0%BA_2011.pdf.
2. Introduction to Genetic Algorithms — Including Example Code [Електронний ресурс] // Medium. – 2017. – Режим доступу до ресурсу: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.
3. Genetic Algorithms [Електронний ресурс] // geeksforgeeks. – 2023. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/genetic-algorithms/.
4. Dr T. K. Study of Various Mutation Operators in Genetic Algorithms / Dr Tapas Kumar, 2014. – 3 с.

ДОДАТКИ

Додаток А. Тексти програмного коду

```
using ConsoleApp1.Crossovers;
using ConsoleApp1.Fabrics;
using ConsoleApp1.GeneticModels;

namespace ConsoleApp1;

class GeneticAlgorithm<T> where T : class, IGeneticModel
{
    private const int PopulationSize = 1000;
    private const int MaxGenerations = 300;
    private const double CrossoverProbability = 0.7;
    private const double MutationProbability = 0.3;

    private readonly IGeneticModelFactory<T> _geneticModelFactory;
    private readonly ICrossoverLogic<T> _crossoverLogic;

    public GeneticAlgorithm(IGeneticModelFactory<T> geneticModelFactory,
ICrossoverLogic<T> crossoverLogic)
    {
        _geneticModelFactory = geneticModelFactory;
        _crossoverLogic = crossoverLogic;
    }

    public Individual<T> Run()
    {
        var population = InitializePopulation();
```

```
        Individual<T> bestIndividual = null;

        for (int generation = 0; generation < MaxGenerations; generation++)
        {
            EvaluatePopulation(population);
            var currentBest = population.OrderByDescending(i => i.Fitness).First();
            Console.WriteLine($"Generation {generation}: Best Fitness = {currentBest.Fitness}");

            if (bestIndividual == null || currentBest.Fitness > bestIndividual.Fitness) bestIndividual = currentBest;

            var selectedParents = TournamentSelection(population);
            population = Crossover(selectedParents);
            Mutate(population);
        }

        Console.WriteLine($"Best Fitness: {bestIndividual.Fitness}");

        return bestIndividual;
    }

    private List<Individual<T>> InitializePopulation()
    {
        var population = new List<Individual<T>>();

        for (var i = 0; i < PopulationSize; i++)
        {
```

```
            Individual<T> individual = new(_geneticModelFactory.CreateRandomModel());
            population.Add(individual);
        }

        return population;
    }

    private void EvaluatePopulation(List<Individual<T>> population)
    {
        foreach (var individual in population)
        {
            individual.GeneticModel.Simulate();
            individual.Fitness = individual.GeneticModel.CalculateProfit();
        }
    }

    private List<Individual<T>> TournamentSelection(List<Individual<T>> population)
    {
        var selectedParents = new List<Individual<T>>();

        for (int i = 0; i < PopulationSize; i++)
        {
            var contestant1 = population[Random.Shared.Next(population.Count)];
            var contestant2 = population[Random.Shared.Next(population.Count)];
            var selectedParent = (contestant1.Fitness > contestant2.Fitness) ? contestant1 : contestant2;
            selectedParents.Add(selectedParent);
```

```
    }

    return selectedParents;
}

private List<Individual<T>> Crossover(List<Individual<T>> parents)
{
    var offspring = new List<Individual<T>>();

    for (int i = 0; i < PopulationSize; i += 2)
    {
        if (Random.Shared.NextDouble() < CrossoverProbability)
        {
            var parent1 = parents[i];
            var parent2 = parents[i + 1];
            offspring.Add(CreateChildModel(parent1, parent2));
            offspring.Add(CreateChildModel(parent2, parent1));
        }
        else
        {
            offspring.Add(parents[i]);
            offspring.Add(parents[i + 1]);
        }
    }

    return offspring;
}
```

```
    private Individual<T> CreateChildModel(Individual<T> parent1, Individual<T> parent2)
    {
        return new Individual<T>(_crossoverLogic.Crossover(parent1.GeneticModel, parent2.GeneticModel));
    }

    private void Mutate(List<Individual<T>> population)
    {
        foreach (var individual in population)
        {
            if (Random.Shared.NextDouble() < MutationProbability)
            {
                individual.GeneticModel.Mutate();
            }
        }
    }
}
```

```
using ConsoleApp1.GeneticModels;

namespace ConsoleApp1;

public class Individual<T> where T: IGeneticModel
{
    public T GeneticModel { get; set; }
    public double Fitness { get; set; }

    public Individual(T geneticModel)
```

```
    {
        GeneticModel = geneticModel;
        Fitness = 0;
    }
}

namespace ConsoleApp1.GeneticModels;

public interface IGeneticModel
{
    void Mutate();
    void Simulate();
    double CalculateProfit();
}

using ConsoleApp1.GeneticModels;

namespace ConsoleApp1.Crossovers;

public interface ICrossoverLogic<T> where T : IGeneticModel
{
    T Crossover(T parent1, T parent2);
}

using ConsoleApp1.GeneticModels;

namespace ConsoleApp1.Fabrics;

public interface IGeneticModelFactory<T> where T: IGeneticModel
```

```
{
    T CreateRandomModel();
}

using MassServiceModeling.Elements;
using MassServiceModeling.Model;
using MassServiceModeling.NextElement;

namespace ConsoleApp1.GeneticModels;

public class Model1 : IGeneticModel
{
    public enum Names
    {
        SubProcessCount1, MaxQueue1,
        SubProcessCount2, MaxQueue2,
        SubProcessCount3, MaxQueue3
    }

    private Model _model;
    public double SimulationTime;
    public Dictionary<Names, int> Data { get;}

    public Model1(Dictionary<Names, int> data, double simulationTime)
    {
        Data = data;
        _model = CreateModel(Data);
        SimulationTime = simulationTime;
    }
```

```
    public void Mutate()
    {
        while (true)
        {
            var parameterIndex = Random.Shared.Next(Data.Count);
            var adder = Random.Shared.Next(-1, 1);
            Data[Enum.GetValues<Names>()[parameterIndex]] += adder;
            if (DataIsValid(Data)) break;
            Data[Enum.GetValues<Names>()[parameterIndex]] -= adder;
        }

        UpdateSMOModel();
    }

    public void Simulate() => _model.Simulate(SimulationTime, printResult: false);
    private void UpdateSMOModel() => _model = CreateModel(Data);

    public static Model CreateModel(Dictionary<Names, int> data)
    {
        Create create = new(delay: 1);
        Process process1 = new(delay: 50, data[Names.SubProcessCount1], "Process1",
data[Names.MaxQueue1]);
        Process process2 = new(delay: 7, data[Names.SubProcessCount2], "Process2",
data[Names.MaxQueue2]);
        Process process3 = new(delay: 12, data[Names.SubProcessCount3], "Process3",
data[Names.MaxQueue3]);

        var container = new NextElementsContainerByQueuePriority();
```

```
        container.AddNextElement(process1, 1);
        container.AddNextElement(process2, 2);
        create.NextElementsContainer = container;
        process2.NextElementsContainer = new NextElementContainer(process3);

        return new Model(new List<Element> { create, process1, process2, process3 });
    }

    public double CalculateProfit()
    {
        return (Data[Names.SubProcessCount1] + Data[Names.SubProcessCount2] +
Data[Names.SubProcessCount3]) * -1 +
                (Data[Names.MaxQueue1] + Data[Names.MaxQueue2] +
Data[Names.MaxQueue3]) * -100 +
                _model.StatisticHelper.FailurePercent * -1000;

        return Data[Names.SubProcessCount1] * -8 +
                Data[Names.MaxQueue1] * -100 +
                Data[Names.SubProcessCount2] * -10 +
                Data[Names.MaxQueue2] * -100 +
                Data[Names.SubProcessCount3] * -7 +
                Data[Names.MaxQueue3] * -100 +
                _model.StatisticHelper.FailurePercent * -1000;
    }

    public bool DataIsValid(Dictionary<Names, int> data)
    {
        for (var i = 0; i < data.Count; i++)
        {
```

```
            if (i % 2 == 0 && data[(Names)i] <= 0) return false;
            if (i % 2 == 1 && data[(Names)i] < 0) return false;
        }

        return true;
    }

    public override string ToString()
    {
        return $"SubProcessCount1: {Data[Names.SubProcessCount1]}\n MaxQueue1:
{Data[Names.MaxQueue1]}\n" +
               $"SubProcessCount2:  {Data[Names.SubProcessCount2]}\n  MaxQueue2:
{Data[Names.MaxQueue2]}\n" +
               $"SubProcessCount3:  {Data[Names.SubProcessCount3]}\n  MaxQueue3:
{Data[Names.MaxQueue3]}\n" +
               $"FailurePercent: {_model.StatisticHelper.FailurePercent}\n" +
               $"Fitness: {CalculateProfit()}";
    }
}

using ConsoleApp1.GeneticModels;

namespace ConsoleApp1.Fabrics;

public class Model1Factory : IGeneticModelFactory<Model1>
{
    public Model1 CreateRandomModel()
    {
        var data = new Dictionary<Model1.Names, int>
```

```
        {
            { Model1.Names.SubProcessCount1, Random.Shared.Next(1, 30) },
            { Model1.Names.MaxQueue1, Random.Shared.Next(0, 30) },
            { Model1.Names.SubProcessCount2, Random.Shared.Next(1, 30) },
            { Model1.Names.MaxQueue2, Random.Shared.Next(0, 30) },
            { Model1.Names.SubProcessCount3, Random.Shared.Next(1, 30) },
            { Model1.Names.MaxQueue3, Random.Shared.Next(0, 30) },
        };
        return new Model1(data, 1000);
    }
}

using ConsoleApp1.GeneticModels;

namespace ConsoleApp1.Crossovers;

public class Model1Crossover : ICrossoverLogic<Model1>
{
    public Model1 Crossover(Model1 parent1, Model1 parent2)
    {
        Dictionary<Model1.Names, int> childData = new();
        for (var i = 0; i < parent1.Data.Count; i++)
            if (i < parent1.Data.Count / 2)
                childData[Enum.GetValues<Model1.Names>()[i]]                    =
parent1.Data[Enum.GetValues<Model1.Names>()[i]];
            else
                childData[Enum.GetValues<Model1.Names>()[i]]                    =
parent2.Data[Enum.GetValues<Model1.Names>()[i]];
```

```
        return new Model1(childData, parent1.SimulationTime);
    }
}

using MassServiceModeling.Items;

namespace ConsoleApp1.GeneticModels.Model2HospitalElements;

public class Client : Item
{
    public ClientType ClientType { get; set; }
    public double RegistrationTime { get; set; }

    public Client(string name, ClientType clientType, double registrationTime, double
startTime) : base(startTime)
    {
        Name = name;
        ClientType = clientType;
        RegistrationTime = registrationTime;
    }
}

public enum ClientType
{
    Chamber,
    NotExamined,
    Lab
}
```

```
using DistributionRandomizer.DelayRandomizers;
using MassServiceModeling.Elements;
using MassServiceModeling.Items;
using MassServiceModeling.TimeClasses;

namespace ConsoleApp1.GeneticModels.Model2HospitalElements;

public class CreateClient : Create
{
   public CreateClient(Randomizer randomizer, string name) : base(randomizer, name)
{}

   protected override Item CreateItem()
   {
      var number = new Random().NextDouble();
      if (number <= 0.5) return new Client("ChamberClient", ClientType.Chamber, 15,
Time.Curr);
      if (number <= 0.6) return new Client("NotExaminedChamberClient",
ClientType.NotExamined, 40, Time.Curr);
      return new Client("LabClient", ClientType.Lab, 30, Time.Curr);
   }
}

using DistributionRandomizer.DelayRandomizers;
using MassServiceModeling.Elements;
using MassServiceModeling.Items;
using MassServiceModeling.NextElement;

namespace ConsoleApp1.GeneticModels.Model2HospitalElements;
```

```
public class DoctorProcess : Process
{
   public DoctorProcess(int doctorsCount, string name, string subProcessName, int maxQueue = 2147483647)
      : base(new UniformRandomizer(-5, 5), doctorsCount, name, maxQueue, subProcessName)
   {
      Queue = new ClientsQueue(maxQueue);
   }
   protected override double GetDelay()
   {
      return (Item as Client).RegistrationTime + Randomizer.GenerateDelay();
   }
}

internal class ClientsQueue : ItemsQueue
{
   public ClientsQueue(int limit) : base(limit) {}
   public override Item GetItem()
   {
      var chamberClient = Queue.Find(x => (x as Client).ClientType == ClientType.Chamber);
      if (chamberClient != null)
      {
         Queue.Remove(chamberClient);
         return chamberClient;
      }
      return base.GetItem();
```

```
    }
}

public class NextAfterDoctor : NextElementsContainer
{
    private Process NextAttendantProcess { get; }
    private Process NextWayToLab { get; }
    private DoctorProcess _doctorProcess;

    public NextAfterDoctor(DoctorProcess doctorProcess, Process
nextAttendantProcess, Process nextWayToLab)
    {
        _doctorProcess = doctorProcess;
        NextAttendantProcess = nextAttendantProcess;
        NextWayToLab = nextWayToLab;
    }

    protected override Element GetNextElement()
    {
        var client = _doctorProcess.Item as Client;
        return client.ClientType == ClientType.Chamber ? NextAttendantProcess :
NextWayToLab;
    }
}

using DistributionRandomizer.DelayRandomizers;
using MassServiceModeling.Elements;

namespace ConsoleApp1.GeneticModels.Model2HospitalElements;
```

```
public class LabAssistanceProcess : Process
{
   public LabAssistanceProcess(Randomizer randomizer, int assistanceCount, string name, string subProcessName, int maxQueue = 2147483647) :
       base(randomizer, assistanceCount, name, maxQueue, subProcessName) {}

   protected override void NextElementsContainerSetup()
   {
      if (Item is Client { ClientType: ClientType.NotExamined } client)
      {
         client.ClientType = ClientType.Chamber;
         client.RegistrationTime = 15;
         client.Name = "ChamberClient";
      }
      else NextElementsContainer = null;
   }
}

using ConsoleApp1.GeneticModels.Model2HospitalElements;
using DistributionRandomizer.DelayRandomizers;
using MassServiceModeling.Elements;
using MassServiceModeling.Model;
using MassServiceModeling.NextElement;

namespace ConsoleApp1.GeneticModels;

public class Model2 : IGeneticModel
{
```

```
public enum Names
{
    DoctorsCount,
    DoctorsQueue,
    AttendantsCount,
    AttendantsQueue,
    RegistryWorkersCount,
    RegistryQueue,
    AssistantsCount,
    AssistantsQueue,
}

private Model _model;
public double SimulationTime;
public Dictionary<Names, int> Data { get; }

public Model2(Dictionary<Names, int> data, double simulationTime)
{
    Data = data;
    _model = CreateModel(Data);
    SimulationTime = simulationTime;
}

public static Model CreateModel(Dictionary<Names, int> data)
{
    CreateClient patients = new(new ExponentialRandomizer(15), "Patient");
    DoctorProcess doctors = new(doctorsCount: data[Names.DoctorsCount],
"Doctors", "Doctor", data[Names.DoctorsQueue]);
```

```
Process attendants = new(new UniformRandomizer(3, 8), data[Names.AttendantsCount], "Attendants", data[Names.AttendantsQueue], "Attendant");
Process fromHospitalToLab = new(new UniformRandomizer(2, 5), 25, name: "WayToLab");
Process labRegistry = new(new ErlangRandomizer(4.5, 3), data[Names.RegistryWorkersCount], "Registry", data[Names.RegistryQueue], "RegistryWorker");
LabAssistanceProcess labAssistants = new(new ErlangRandomizer(4, 2), data[Names.AssistantsCount], "Assistants", "Assistant", data[Names.AssistantsQueue]);
Process fromLabToHospital = new(new UniformRandomizer(2, 5), 25, name: "WayToHospital");

patients.NextElementsContainer = new NextElementContainer(doctors);
doctors.NextElementsContainer = new NextAfterDoctor(doctors, attendants, fromHospitalToLab);
fromHospitalToLab.NextElementsContainer = new NextElementContainer(labRegistry);
labRegistry.NextElementsContainer = new NextElementContainer(labAssistants);
labAssistants.NextElementsContainer = new NextElementContainer(fromLabToHospital);
fromLabToHospital.NextElementsContainer = new NextElementContainer(doctors);

return new Model(new List<Element>()
    { patients, doctors, attendants, fromHospitalToLab, labRegistry, labAssistants, fromLabToHospital });
```

```
}

public void Simulate() => _model.Simulate(SimulationTime, printResult: false);
private void UpdateSMOModel() => _model = CreateModel(Data);


public void Mutate()
{
    while (true)
    {
        var parameterIndex = Random.Shared.Next(Data.Count);
        var adder = Random.Shared.Next(-1, 1);
        Data[Enum.GetValues<Names>()[parameterIndex]] += adder;
        if (DataIsValid(Data)) break;
        Data[Enum.GetValues<Names>()[parameterIndex]] -= adder;
    }

    UpdateSMOModel();
}

private bool DataIsValid(Dictionary<Names, int> data)
{
    for (int i = 0; i < data.Count; i++)
    {
        if (i % 2 == 0 && data[(Names)i] <= 0) return false;
        if (i % 2 == 1 && data[(Names)i] < 0) return false;
    }

    return true;
```

```
    }

    public double CalculateProfit()
    {
        var queueMax = -100;
        return Data[Names.DoctorsCount] * -100 +
               Data[Names.DoctorsQueue] * -1 +
               Data[Names.AttendantsCount] * -100 +
               Data[Names.AttendantsQueue] * -1 +
               Data[Names.RegistryWorkersCount] * -100 +
               Data[Names.RegistryQueue] * -1 +
               Data[Names.AssistantsCount] * -100 +
               Data[Names.AssistantsQueue] * -1 +
               _model.StatisticHelper.AverageItemTimeInSystem * -10 +
               _model.StatisticHelper.FailurePercent * -10000;
    }

    public override string ToString()
    {
        return $"DoctorsCount: {Data[Names.DoctorsCount]}\n DoctorsQueue:
{Data[Names.DoctorsQueue]}\n" +
               $"AttendantsCount: {Data[Names.AttendantsCount]}\n AttendantsQueue:
{Data[Names.AttendantsQueue]}\n" +
               $"RegistryWorkersCount: {Data[Names.RegistryWorkersCount]}\n
RegistryQueue: {Data[Names.RegistryQueue]}\n" +
               $"AssistantsCount: {Data[Names.AssistantsCount]}\n AssistantsQueue:
{Data[Names.AssistantsQueue]}\n" +
               $"AverageItemTimeInSystem:
{_model.StatisticHelper.AverageItemTimeInSystem}\n" +
```

```
            $"FailurePercent: {_model.StatisticHelper.FailurePercent}\n" +
            $"Fitness: {CalculateProfit()}";
    }
}

using ConsoleApp1.GeneticModels;

namespace ConsoleApp1.Crossovers;

public class Model2Crossover : ICrossoverLogic<Model2>
{
    public Model2 Crossover(Model2 parent1, Model2 parent2)
    {
        Dictionary<Model2.Names, int> childData = new();
        for (var i = 0; i < parent1.Data.Count; i++)
            if (i < parent1.Data.Count / 2)
                childData[Enum.GetValues<Model2.Names>()[i]]                    =
parent1.Data[Enum.GetValues<Model2.Names>()[i]];
            else
                childData[Enum.GetValues<Model2.Names>()[i]]                    =
parent2.Data[Enum.GetValues<Model2.Names>()[i]];

        return new Model2(childData, parent1.SimulationTime);
    }
}

using ConsoleApp1.GeneticModels;

namespace ConsoleApp1.Fabrics;
```

```
public class Model2Factory : IGeneticModelFactory<Model2>
{
    public Model2 CreateRandomModel()
    {
        var data = new Dictionary<Model2.Names, int>()
        {
            { Model2.Names.DoctorsCount, Random.Shared.Next(1, 5) },
            { Model2.Names.DoctorsQueue, Random.Shared.Next(0, 100) },
            { Model2.Names.AttendantsCount, Random.Shared.Next(1, 5) },
            { Model2.Names.AttendantsQueue, Random.Shared.Next(0, 100) },
            { Model2.Names.RegistryWorkersCount, Random.Shared.Next(1, 5) },
            { Model2.Names.RegistryQueue, Random.Shared.Next(0, 100) },
            { Model2.Names.AssistantsCount, Random.Shared.Next(1, 5) },
            { Model2.Names.AssistantsQueue, Random.Shared.Next(0, 100) },
        };
        return new Model2(data, 3000);
    }
}

using DistributionRandomizer.DelayRandomizers;
using MassServiceModeling.Items;
using MassServiceModeling.Models;
using MassServiceModeling.NextElement;
using MassServiceModeling.Printers;
using MassServiceModeling.Statistics;
using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.Elements;
```

```
public abstract class Element
{
    // Non-static attributes
    public bool IsWorking { get; protected set; }
    public Item? Item { get; protected set; }
    public double NextT = double.MaxValue;
    public double Delay;

    // Static attributes
    public NextElementsContainer? NextElementsContainer;
    public Model? Model { get; set; }
    public ElementStatisticHelper BaseStatistic = new();
    public IPrinter Print { get; protected init; }
    public Randomizer Randomizer { get; }

    // Self-static
    public string Name { get; }
    private static int _nextId;
    public int Id { get; } = _nextId++;

    protected Element(Randomizer randomizer, string name)
    {
        Name = name == "" ? $"{GetElementDefaultName()}_{Id}" : name;
        Print = new ElementPrinter(this);
        Randomizer = randomizer;
    }

    public virtual void InAct(Item item)
```

```
    {
        IsWorking = true;
        DoInActStatistics();
        SetItem(item);
    }

    public virtual void OutAct()
    {
        IsWorking = false;
        DoOutActStatistics();
        NextElementsContainer?.InAct(Item ?? throw new
InvalidOperationException());
        Item = null;
        UpdateNextT();
    }

    public virtual void DoStatistics(double delta) => BaseStatistic.WorkTime +=
IsWorking ? delta : 0;

    protected virtual double GetDelay() => Delay = Randomizer.GenerateDelay();

    private void DoInActStatistics() => BaseStatistic.InAct();

    private void DoOutActStatistics()
    {
        if (Item is null) {throw new InvalidOperationException();}
        IPrinter.CurrentItem = Item;
        Model!.AddItemTimeInSystem(Time.Curr - Item.StartTime);
        BaseStatistic.OutAct();
```

```
    }

    protected abstract void UpdateNextT();

    protected abstract void SetItem(Item item);

    protected abstract string GetElementDefaultName();
}

using DistributionRandomizer.DelayRandomizers;
using MassServiceModeling.Items;
using MassServiceModeling.Printers;
using MassServiceModeling.Statistics;
using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.Elements;

public class Create : Element
{
    public CreateStatisticHelper CreateStatistic = new();
    public Create(Randomizer randomizer, string name = "") : base(randomizer, name)
    {
        NextT = Time.Start;
        Print = new CreatePrinter(this);
    }

    public override void OutAct()
    {
        Item = CreateItem();
```

```
        base.OutAct();
    }

    public Create(double delay = 1.0, string name = "CREATE") : this(new
ExponentialRandomizer(delay), name) {}

    protected virtual Item CreateItem() => new(Time.Curr);

    protected override void SetItem(Item item) => Item = item;

    protected override string GetElementDefaultName() => "CREATE";

    protected override void UpdateNextT() => NextT = Time.Curr + GetDelay();
}

using DistributionRandomizer.DelayRandomizers;
using MassServiceModeling.Items;
using MassServiceModeling.Printers;
using MassServiceModeling.Statistics;
using MassServiceModeling.SubProcesses;
using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.Elements;

public class Process : Element
{
    public event Action? OnQueueChanged;
    public ItemsQueue Queue;
    public SubProcessesContainer SubProcesses = new();
```

```
    public ProcessStatisticHelper ProcessStatistic = new();

    public Process(Randomizer randomizer, int subProcessCount = 1, string name = "",
int maxQueue = int.MaxValue, String subProcessName = "")
        : base(randomizer, name)
    {
        for (var i = 0; i < subProcessCount; i++) SubProcesses.Add(new SubProcess(i,
subProcessName));
        Queue = new ItemsQueue(maxQueue);
        NextT = double.MaxValue;
        Print = new ProcessPrinter(this);
    }

    public Process(double delay = 1.0, int subProcessCount = 1, string name = "", int
maxQueue = int.MaxValue, String subProcessName = "")
        : this(new ExponentialRandomizer(delay), subProcessCount, name, maxQueue,
subProcessName) {}

    public override void InAct(Item item)
    {
        base.InAct(item);
        UpdateNextT();
    }

    public override void OutAct()
    {
        foreach (var subProcess in SubProcesses.ForOutAct)
        {
            Item = subProcess.OutAct();
```

```
        var nextElements = NextElementsContainer;
        NextElementsContainerSetup();
        base.OutAct();
        NextElementsContainer = nextElements;

        if (SubProcesses.WorkingCount > 0) IsWorking = true;
        if (Queue.Length > 0)
        {
            IsWorking = true;
            Item = Queue.GetItem();
            OnQueueChanged?.Invoke();
            subProcess.InAct(Time.Curr + GetDelay(), Item);
        }
    }
    UpdateNextT();
}

public override void DoStatistics(double delta)
{
    base.DoStatistics(delta);
    foreach (var subProcess in SubProcesses.All) subProcess.DoStatistics(delta);
    ProcessStatistic.MeanQueueAllTime += Queue.Length * delta;
}

public static void TryChangeQueueForLastItem(Process from, Process to)
{
    if (!ItemsQueue.TrySwapLast(from.Queue, to.Queue)) return;
    from.OnQueueChanged?.Invoke();
```

```
        to.OnQueueChanged?.Invoke();
    }

    protected override void SetItem(Item item)
    {
        if (SubProcesses.WorkingCount < SubProcesses.Count)
        {
            Item = item;
            SubProcesses.Free.InAct(Time.Curr + GetDelay(), item);
        }
        else
        {
            if (Queue.TryAdd(item)) OnQueueChanged?.Invoke();
            else ProcessStatistic.Failure++;
        }
    }

    protected override void UpdateNextT() => NextT = SubProcesses.All.Min(s => s.NextT);
    protected override string GetElementDefaultName() => "PROCESS";
    protected virtual void NextElementsContainerSetup() {}
}

namespace MassServiceModeling.Items;

public class Item
{
    private static int _id;
    public int Id { get; } = _id++;
```

```
    public string Name { get; set;  } = "";
    public double StartTime { get; }

    public Item(double startTime)
    {
        StartTime = startTime;
    }
}

namespace MassServiceModeling.Items;

public class ItemsQueue
{
    public int Limit { get; }
    public int Length => Queue.Count;
    protected List<Item> Queue { get; } = new();

    public ItemsQueue(int limit = int.MaxValue)
    {
        Limit = limit;
    }

    public bool TryAdd(Item item)
    {
        if (Length >= Limit) return false;
        Queue.Add(item);
        return true;
    }
```

```
    public static bool TrySwapLast(ItemsQueue from, ItemsQueue to)
    {
        if (from.Length <= 0 || to.Length >= to.Limit) return false;
        to.Queue.Add(from.Queue[from.Length - 1]);
        from.Queue.RemoveAt(from.Length - 1);
        return true;
    }

    public virtual Item GetItem()
    {
        var item = Queue[0];
        Queue.RemoveAt(0);
        return item;
    }
}

using MassServiceModeling.Elements;
using MassServiceModeling.Printers;
using MassServiceModeling.Statistics;
using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.Models;

public class Model
{
    // Elements collections
    public List<Create> Creates => Elements.OfType<Create>().ToList();
    public List<Process> Processes => Elements.OfType<Process>().ToList();
    protected readonly List<Element> Elements;
```

```
    public event Action? OnNextElementStarted;
    public ModelStatisticHelper StatisticHelper;
    protected bool InitialStateAccessed { get; }
    protected double NextT;
    private int _event;

    public Model(List<Element> elements, bool initialStateIsNeeded = false)
    {
        Elements = elements;
        StatisticHelper = new ModelStatisticHelper(this);
        InitialStateAccessed = !initialStateIsNeeded;
        Elements.ForEach(e => e.Model = this);
    }

    public virtual void Simulate(double time, double startTime = 0, bool printSteps =
false, bool printResult = true)
    {
        Time.SetStart(startTime);

        while (Time.Curr < time)
        {
            DefineNextEvent();
            if (InitialStateAccessed) DoStatistics();
            else OnNextElementStarted?.Invoke();
            Time.ShiftCurr(NextT);
            OutActForFinished();
            if (printSteps)
            {
```

```
                IPrinter.PrintCurrent(Elements[_event]);
                IPrinter.Info(Elements, Elements[_event]);
            }
        }
        if (printResult) IPrinter.Result(Elements);
    }

    protected virtual void DoStatistics()
    {
        Elements.ForEach(e => e.DoStatistics(Time.Delta(NextT)));
        StatisticHelper.AverageItemsCountAllTime += Processes.Sum(p =>
p.IsWorking ? p.Queue.Length + 1 : 0) * Time.Delta(NextT);
    }

    public void AddItemTimeInSystem(double timeInSystem)
    {
        StatisticHelper.FinishedItemsCount++;
        StatisticHelper.AllFinishedItemsTimeInSystem += timeInSystem;
    }

    private void DefineNextEvent()
    {
        NextT = double.MaxValue;
        for (var i = 0; i < Elements.Count; i++)
        {
            if (!(Elements[i].NextT < NextT)) continue;
            NextT = Elements[i].NextT;
            _event = i;
        }
```

```
    }

    private void OutActForFinished()
    {
        foreach (var element in Elements.Where(element => element.NextT == Time.Curr))
            element.OutAct();
    }
}

using MassServiceModeling.Items;
using MassServiceModeling.Printers;
using MassServiceModeling.Statistics;
using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.SubProcesses;

public class SubProcess
{
    // Dynamic attributes
    public double NextT { get; private set; } = double.MaxValue;
    public Item? Item { private set; get; }
    public double Delay { get; private set; }
    public bool IsWorking { get; private set; }

    // Static attributes
    public string Name { get; }
    public SubProcessStatisticHelper StatisticHelper;
    public SubProcessPrinter Printer { get; private init; }
```

```
public SubProcess(int subProcessId, string name)
{
    Name = name == "" ? "SubProcess" : name;
    Name = $"{Name}_{subProcessId}";
    StatisticHelper = new SubProcessStatisticHelper();
    Printer = new SubProcessPrinter(this);
}

public void InAct(double nextT, Item item)
{
    IsWorking = true;
    StatisticHelper.Quantity++;
    NextT = nextT;
    Delay = nextT - Time.Curr;
    Item = item;
}

public Item OutAct()
{
    IsWorking = false;
    NextT = double.MaxValue;
    return Item!;
}

public void DoStatistics(double delta)
{
    StatisticHelper.WorkTime += IsWorking? delta : 0;
}
```

```
}

using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.SubProcesses;

public class SubProcessesContainer
{
    public int Count => Container.Count;
    public int WorkingCount => Container.Count(s => s.IsWorking);

    public List<SubProcess> All => Container;
    public List<SubProcess> ForOutAct => Container.Where(s => s.NextT <= Time.Curr && s.IsWorking).ToList();
    public SubProcess Free => Container.First(s => !s.IsWorking);

    protected List<SubProcess> Container { get; } = new();

    public void Add(SubProcess subProcess) => Container.Add(subProcess);
}

namespace MassServiceModeling.TimeClasses;

public static class Time
{
    public static double Curr;
    public static double Start { get; private set; }
    public static double All => Curr - Start;
```

```
    public static void ShiftCurr(double next) => Curr = next;
    public static double Delta(double next) => next - Curr;
    public static void SetStart(double start) => Curr = Start = start;
}

using MassServiceModeling.Elements;
using MassServiceModeling.Items;

namespace MassServiceModeling.NextElement;

public class NextElement
{
    public readonly Element Element;
    public readonly double Probability;
    public NextElement(Element element, double probability = 1)
    {
        Element = element;
        Probability = probability;
    }
}

public abstract class NextElementsContainer
{
    public void InAct(Item item) => GetNextElement().InAct(item);
    protected abstract Element GetNextElement();
}

using MassServiceModeling.Elements;
```

```
namespace MassServiceModeling.NextElement;

public class NextElementContainer : NextElementsContainer
{
    private readonly Element _nextElement;

    public NextElementContainer(Element nextElement)
    {
        _nextElement = nextElement;
    }

    protected override Element GetNextElement() => _nextElement;
}

using MassServiceModeling.Elements;
namespace MassServiceModeling.NextElement;

public class NextElementsContainerByQueuePriority : NextElementsContainer
{
    private Dictionary<Process, int> _nextElements = new();
    public void AddNextElement(Process process, int ascendingPriority) =>
_nextElements.Add(process, ascendingPriority);

    protected override Element GetNextElement()
    {
        var elementsWithMinQueueLength = GetElementsWithMinQueueLength();
        return elementsWithMinQueueLength.MinBy(e => e.Value).Key;
    }
```

```
    private Dictionary<Process, int> GetElementsWithMinQueueLength()
    {
        var minQueueLength = _nextElements.Min(e => e.Key.Queue.Length);
        var nextElementsWithMinQueueLength = new Dictionary<Process, int>();
        foreach (var (process, priority) in _nextElements)
            if (process.Queue.Length == minQueueLength)
                nextElementsWithMinQueueLength.Add(process, priority);
        return nextElementsWithMinQueueLength;
    }
}

using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.Statistics;

public class ElementStatisticHelper
{
    public double WorkTime { get; set; }
    public int InActQuantity { get; private set; }
    public int OutActQuantity { get; private set; }
    public double TotalTimeBetweenInActs { get; private set; }
    public double TotalTimeBetweenOutActs { get; private set; }
    private double? _lastInActTime;
    private double? _lastOutActTime;

    public void InAct()
    {
        InActQuantity++;
        TotalTimeBetweenInActs += Time.Curr - _lastInActTime ?? 0;
```

```
        _lastInActTime = Time.Curr;
    }

    public void OutAct()
    {
        OutActQuantity++;
        TotalTimeBetweenOutActs += Time.Curr - _lastOutActTime ?? 0;
        _lastOutActTime = Time.Curr;
    }
}

using MassServiceModeling.Models;
using MassServiceModeling.TimeClasses;

namespace MassServiceModeling.Statistics;

public class ModelStatisticHelper
{
    // Items Count
    public int Quantity => _model.Creates.Sum(e => e.BaseStatistic.OutActQuantity);
    public double AverageItemsCount => AverageItemsCountAllTime / Time.All;
    public double AverageItemsCountAllTime;

    // Failure
    public int FailureQuantity => _model.Processes.Sum(e => e.ProcessStatistic.Failure);
    public double FailurePercent => (double)FailureQuantity / Quantity * 100;

    // ItemsTime in system
```

```
    public double AverageItemTimeInSystem => AllFinishedItemsTimeInSystem / FinishedItemsCount;
    public int FinishedItemsCount { get; set; }
    public double AllFinishedItemsTimeInSystem { get; set; }

    private Model _model;

    public ModelStatisticHelper(Model model) => _model = model;
}

namespace MassServiceModeling.Statistics;

public class ProcessStatisticHelper : ElementStatisticHelper
{
    public int Failure { get; set; }
    public double MeanQueueAllTime { get; set; }
}

namespace MassServiceModeling.Statistics;

public class SubProcessStatisticHelper
{
    public int Quantity { get; set; }
    public double WorkTime { get; set; }
}
```

Додаток Б. Результати експериментів

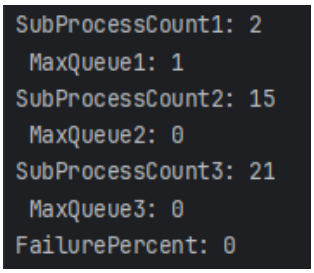

Рисунок 1 – Результати Model1 із вірогідністю схрещування 0.7 та мутації 0.1

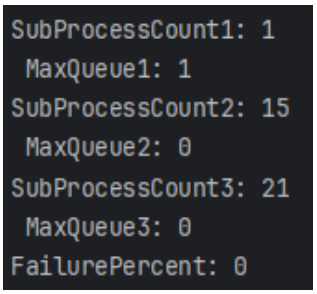

Рисунок 2 – Результати Model1 із вірогідністю схрещування 1.0 та мутації 0.5

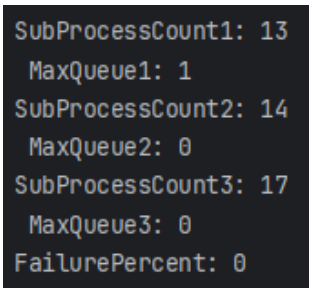

Рисунок 3 – Результати Model1 із вірогідністю схрещування 0.7 та мутації 0.1 з оновленою функцією CalculateProfit

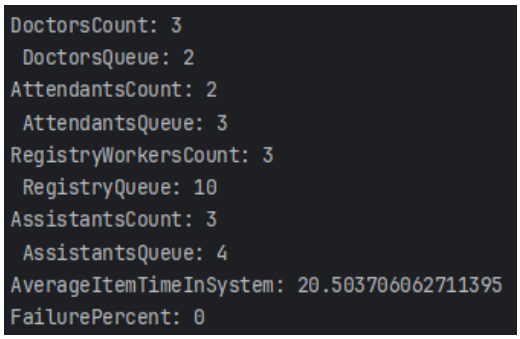

Рисунок 4 – Результати Model2 задача №1, 100 ітерацій та 400 популяція

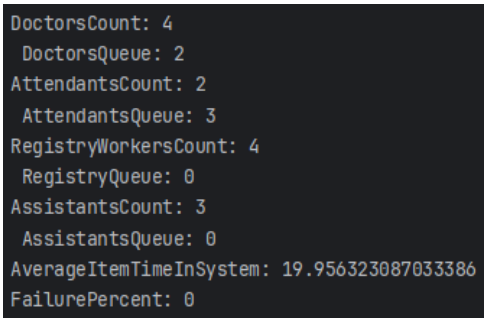

Рисунок 5 – Результати Model2 задача №1, 200 ітерацій та 700 популяція

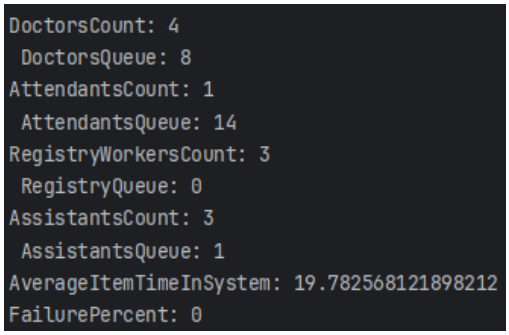

Рисунок 6 – Результати невдалого тесту №2 Model2 задача№1 із коефіцієнтом - 200 для RegistryQueue та AssistantsQueue

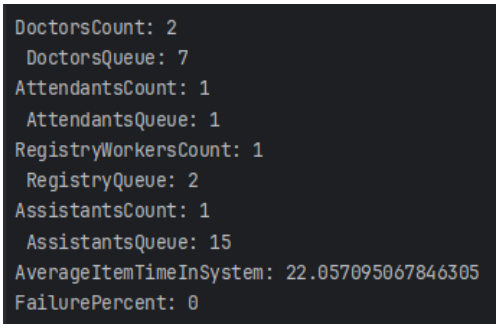

Рисунок 7 – Результати 1-го тесту Model2 задача №2

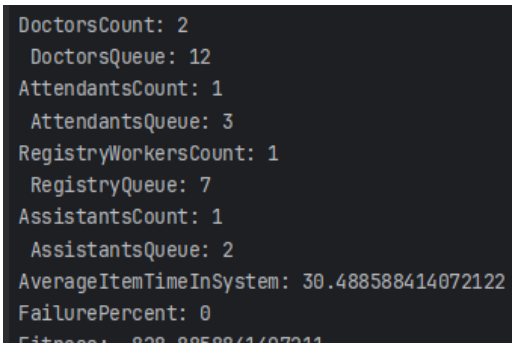

Рисунок 8 – Результати 2-го тесту Model2 задача №2