Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 5 з дисципліни
«Проектування алгоритмів»

„Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2”

Виконав(ла) *ІП-15, Плугатирьов Д.В.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.Н.* _____
(прізвище, ім'я, по батькові)

Київ 2023

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ 3

2 ЗАВДАННЯ 4

3 ВИКОНАННЯ 11

3.1 Покроковий алгоритм 11

3.2 Програмна реалізація алгоритму 13

3.2.1 Вихідний код 13

3.2.2 Приклади роботи 35

3.3 Тестування алгоритму 37

ВИСНОВОК 40

КРИТЕРІЇ ОЦІНЮВАННЯ 41

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи розробки метаевристичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

2 ЗАВДАННЯ

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

– обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);

– зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;

– після цього параметр фіксується і змінюються інші параметри;

– далі повторюємо процедуру спочатку, з першого зафіксованого параметру;

– зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

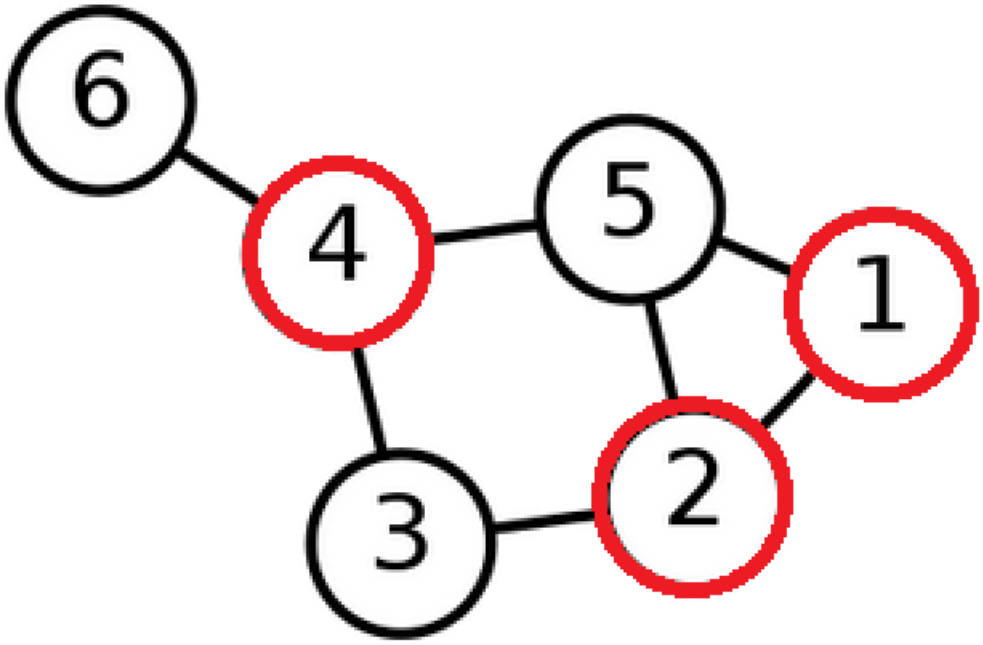
Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

№	**Задача**
1	**Задача про рюкзак** (місткість P=500, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб

	сумарна вага не перевищувала задану, а сумарна цінність була максимальною. Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика.
2	**Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів. **Розглядається симетричний, асиметричний та змішаний варіанти.** В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра. У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу. У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху. Застосування: – доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); – доставка води;

	– моніторинг об'єктів; – поповнення банкоматів готівкою; – збір співробітників для доставки вахтовим методом.
3	**Розфарбовування графа** (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число. Застосування: – розкладу для освітніх установ; – розкладу в спорті; – планування зустрічей, зборів, інтерв'ю; – розклади транспорту, в тому числі - авіатранспорту; – розкладу для комунальних служб;
4	**Задача вершинного покриття** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Вершинне покриття для неорієнтованого графа G = (V, E) - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S. Задача вершинного покриття полягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа). На вході: Граф G = (V, E). Результат: множина C ⊆ V - найменше вершинне покриття графа G.

	 Застосування: – розміщення пунктів обслуговування; – призначення екіпажів на транспорт; – проектування інтегральних схем і конвеєрних ліній.
5	**Задача про кліку** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній. Задача про кліку існує у двох варіантах: у **задачі розпізнавання** потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в **обчислювальному варіанті** потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити). Застосування: – біоінформатика; – електротехніка;
6	**Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але

	не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях. Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший.

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

№	**Алгоритми і досліджувані параметри**
1	**Генетичний алгоритм:** - оператор схрещування (мінімум 3); - мутація (мінімум 2); - оператор локального покращення (мінімум 2).
2	**Мурашиний алгоритм**: – α; – β; – ρ; – L_{min}; – кількість мурах M і їх типи (елітні, тощо…); – маршрути з однієї чи різних вершин.
3	**Бджолиний алгоритм:** – кількість ділянок; – кількість бджіл (фуражирів і розвідників).

Таблиця 2.3 – Варіанти задач і алгоритмів

№	Задачі і алгоритми
1	Задача про рюкзак + Генетичний алгоритм
2	Задача про рюкзак + Бджолиний алгоритм
3	Задача комівояжера (асиметрична мережа) + Генетичний алгоритм
4	Задача комівояжера (симетрична мережа) + Генетичний алгоритм
5	Задача комівояжера (змішана мережа) + Генетичний алгоритм
6	Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм
7	Задача комівояжера (симетрична мережа) + Мурашиний алгоритм
8	Задача комівояжера (змішана мережа) + Мурашиний алгоритм
9	Задача вершинного покриття + Генетичний алгоритм
10	Задача вершинного покриття + Бджолиний алгоритм
11	Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм
12	Задача комівояжера (симетрична мережа) + Бджолиний алгоритм
13	Задача комівояжера (змішана мережа) + Бджолиний алгоритм
14	Розфарбовування графа + Генетичний алгоритм
15	Розфарбовування графа + Бджолиний алгоритм
16	Задача про кліку (задача розпізнавання) + Генетичний алгоритм
17	Задача про кліку (задача розпізнавання) + Бджолиний алгоритм
18	Задача про кліку (обчислювальна задача) + Генетичний алгоритм
19	Задача про кліку (обчислювальна задача) + Бджолиний алгоритм
20	Задача про найкоротший шлях + Генетичний алгоритм
21	Задача про найкоротший шлях + Мурашиний алгоритм
22	Задача про найкоротший шлях + Бджолиний алгоритм
23	Задача про рюкзак + Генетичний алгоритм
24	Задача про рюкзак + Бджолиний алгоритм
25	Задача комівояжера (асиметрична мережа) + Генетичний алгоритм
26	Задача комівояжера (симетрична мережа) + Генетичний алгоритм
27	Задача комівояжера (змішана мережа) + Генетичний алгоритм

28	Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм
29	Задача комівояжера (симетрична мережа) + Мурашиний алгоритм
30	Задача комівояжера (змішана мережа) + Мурашиний алгоритм

3 ВИКОНАННЯ

3.1 Покроковий алгоритм

3.1.1 Основний алгоритм

Змінна	**Призначення**
iterationsCount	К-сть ітерацій алгоритму
i	Лічильник циклу
offspring	Нащадок
maximalProbabilityLevel	Максимально доступне значення ймовірності мутації
mutationProbability	Випадково згенерована ймовірність мутації

1. ПОЧАТОК
2. Згенерувати хромосоми
3. Визначити maximalProbabilityLevel
4. ЦИКЛ ПОКИ i < iterationsCount
    5. Провести схрещення
    6. ЯКЩО offspring == null ТО
        7. continue
    8. Випадково згенерувати mutationProbability
    9. ЯКЩО mutationProbability <= maximalProbabilityLevel ТО
        10. Застосувати мутацію до offspring
    11. Застосувати оператор покращення до offspring
    12. Додати offspring до популяції
    13. Видалити хромосому із найбільшою довжиною між початковою та кінцевою вершинами
    14. i = i + 1
15. КІНЕЦЬ ЦИКЛУ
16. Повернути найкоротший шлях

3.1.2 Алгоритм створення хромосоми

Змінна	Призначення
chromosomeToAdd	Хромосома, котра в процесі створення

1. ПОЧАТОК
2. Створити пусту хромосому chromosomeToAdd
3. Додати до chromosomeToAdd початкову вершину
4. ПОЧАТОК ЦИКЛУ
    5. Отримати всі дуги, протилежні вершини яких можна з’єднати без повторів із останньою доданою вершиною chromosomeToAdd
    6. ЯКЩО к-сть знайдених дуг нульова ТО
        7. Видалити всі вершини із chromosomeToAdd, окрім першої
        8. continue
    9. Отримати випадковий індекс одної зі знайдених дуг
    10. Отримати число вершини із дуги за індексом
    11. Додати знайдений номер вершини до chromosomeToAdd
    12. ЯКЩО існує хромосома із такими самими номерами вершин ТО
        13. Видалити всі номери вершин у chromosomeToAdd, окрім початкового
14. ПОКИ останній номер chromosomeToAdd не дорівнює кінцевому номеру
15. Додати chromosomeToAdd до популяції

Вибір батьків

Під час вибору батьків я надав перевагу пропорційній селекції на противагу вибору кращого в популяції та випадкового батьків. Це збільшує шанси на те, що знайдеться більш оптимальний шлях, адже найкращий у популяції буде лише псувати збагачення нащадків «свіжими» генами.

Оператор схрещування

Мною було використано одноточковий та рівномірний оператори. Судячи із висновків роботи алгоритму, одноточковий краще себе показує на великих графах, а рівномірний – на малих. Оскільки особливістю рівномірного оператору схрещування є його нестабільний випадковий переніс генів то з однієї хромосоми, то з іншої до нащадка без урахування можливості генів мати спільні дуги, через що багато із них не виживають.

Одноточковий же оператор просто схрещує хромосоми за спільною вершиною, якщо така існує з можливістю як набуття, так і втрати генів. Із часом, останнє переважає, що змушує хромосоми приймати однаковий набір генів на великих ітераціях.

Оператор мутації

В якості оператору мутації мною було використано вставку випадкового гена у випадкове місце хромосоми. Це дозволило мені знаходити нові шляхи більш ефективно.

Локальне покращення

В якості локального покращення я у випадкове місце хромосоми вставляв ген, який, на відміну від усіх інших, маючих спільні дуги із його сусідами, зменшував довжину хромосоми найбільше.

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

```
using System.Collections;

namespace ShortestPathProblemLogic
{
    public class PopulationGenerator : IEnumerable<Chromosome>
    {
        private GraphOfSites _graph;
        private List<Chromosome> _chromosomes = new List<Chromosome>();

        public int StartVertexNumber { get; }

        public int EndVertexNumber { get; }

        /// <exception cref="ArgumentOutOfRangeException"></exception>
```

```
        public PopulationGenerator(GraphOfSites graph, int startVertexNumber, int
endVertexNumber)
        {
            _graph = graph;

            if ((startVertexNumber < 1) || (startVertexNumber >= endVertexNumber))
            {
                throw new ArgumentOutOfRangeException(nameof(startVertexNumber));
            }

            if (endVertexNumber > graph.GetVerticesCount())
            {
                throw new ArgumentOutOfRangeException(nameof(endVertexNumber));
            }

            StartVertexNumber = startVertexNumber;
            EndVertexNumber = endVertexNumber;
        }

        public void AddChromosome(Chromosome chromosome)
        {
            _chromosomes.Add(chromosome);
        }

        public void RemoveChromosome(Chromosome chromosome)
        {
            if (chromosome is null)
            {
                throw new ArgumentNullException(nameof(chromosome));
            }

            if (ContainsChromosome(chromosome.GetVerticesNumbers()))
            {

            }

            _chromosomes.Remove(chromosome);
        }

        public bool ContainsChromosome(List<int> verticesNumbers)
        {
            return _chromosomes
                .Exists(c => Enumerable.SequenceEqual(c.GetVerticesNumbers(),
verticesNumbers));
        }

        public void GenerateChromosomes(int chromosomesCount)
        {
            for (var i = 0; i < chromosomesCount; i++)
            {
```

```
                CreateChromosome();
            }
        }

        public void CreateChromosome()
        {
            var chromosomeToAdd = new Chromosome();
            chromosomeToAdd.AddVertexNumber(StartVertexNumber);

            do
            {
                var allowedEdges = _graph
                    .GetEdgesWithSpecifiedVertexExcept(chromosomeToAdd.LastVertexNu
mber(),
                    chromosomeToAdd.GetVerticesNumbers());

                if (allowedEdges.Count == 0)
                {
                    chromosomeToAdd.RemoveVerticesNumbersExceptStart();
                    continue;
                }

                var random = new Random();
                var randomEdgeIndex = random.Next(allowedEdges.Count);
                var vertexNumberToAdd =
(allowedEdges[randomEdgeIndex].FirstVertex.Number
                    != chromosomeToAdd.LastVertexNumber())
                    ? allowedEdges[randomEdgeIndex].FirstVertex.Number
                    : allowedEdges[randomEdgeIndex].SecondVertex.Number;

                chromosomeToAdd.AddVertexNumber(vertexNumberToAdd);

                if (ContainsChromosome(chromosomeToAdd.GetVerticesNumbers()))
                {
                    chromosomeToAdd.RemoveVerticesNumbersExceptStart();
                }
            } while (chromosomeToAdd.LastVertexNumber() != EndVertexNumber);

            _chromosomes.Add(chromosomeToAdd);
    }

        public IEnumerator<Chromosome> GetEnumerator()
        {
            foreach (Chromosome chromosome in _chromosomes)
            {
                yield return chromosome;
            }
        }

        IEnumerator IEnumerable.GetEnumerator()
```

```
        {
            return GetEnumerator();
        }
    }
}
```

```
using System.Collections;
using System.Text;

namespace ShortestPathProblemLogic
{
    public class GraphOfSites : IEnumerable<GraphEdge>
    {
        private List<GraphVertex> _vertices = new List<GraphVertex>();
        private List<GraphEdge> _edges = new List<GraphEdge>();
        private int _nextVertexNumber = 1;

        /// <exception cref="InvalidOperationException"></exception>
        public GraphOfSites(int initialVerticesCount)
        {
            for (var i = 0; i < initialVerticesCount; i++)
            {
                AddVertice();
            }

            GenerateEdgesCovering();
        }

        public GraphVertex GetVertexByNumber(int vertexNumber)
        {
            return _vertices.Find(v => v.Number == vertexNumber);
        }

        public void UpdateEdgeVertexNumber(GraphEdge edgeToUpdate, int
numberToStay, int numberToAssign)
        {
            if (edgeToUpdate.FirstVertex.Number == numberToStay)
            {
                edgeToUpdate.SecondVertex = GetVertexByNumber(numberToAssign);
            }
            else
            {
                edgeToUpdate.FirstVertex = GetVertexByNumber(numberToAssign);
            }
        }

        public List<int> GetOtherVerticesOfEdgesWithSpecifiedVertexNumber(int
vertexNumber)
        {
            return GetEdgesWithSpecifiedVertex(vertexNumber)
```

```
                .Select(e => (e.FirstVertex.Number == vertexNumber)
                ? e.SecondVertex.Number : e.FirstVertex.Number).ToList();
        }

        public List<GraphEdge> GetEdgesWhichConnectVertices(List<int>
verticesNumbers)
        {
            var result = new List<GraphEdge>();

            for (var i = 0; i < verticesNumbers.Count - 1; i++)
            {
                result.Add(FindEdgeByEndsNumbers(verticesNumbers[i],
verticesNumbers[i + 1]));
            }

            return result;
        }

        public List<GraphEdge> GetEdgesWithSpecifiedVertex(int vertexNumber)
        {
            return _edges.FindAll(e => (e.FirstVertex.Number == vertexNumber)
                || (e.SecondVertex.Number == vertexNumber)).ToList();
        }

        public List<GraphEdge> GetEdgesWithSpecifiedVertexExcept(int vertexNumber,
            List<int> exceptionalVerticesNumbers)
        {
            return GetEdgesWithSpecifiedVertex(vertexNumber).Except
                (GetEdgesWithSpecifiedVertex(vertexNumber)
                .Where(e => ((e.FirstVertex.Number == vertexNumber)
                    &&
(exceptionalVerticesNumbers.Contains(e.SecondVertex.Number)))
                    || ((e.SecondVertex.Number == vertexNumber)
                    &&
(exceptionalVerticesNumbers.Contains(e.FirstVertex.Number))))).ToList();
        }

        /// <exception cref="InvalidOperationException"></exception>
        private void GenerateEdgesCovering()
        {
            if (GetVerticesCount() < GraphValidator.MinimalVerticesCount)
            {
                throw new InvalidOperationException("The count of vertices should
be at least 2");
            }

            GenerateMinimalGraphEdgesCovering();

            foreach (GraphVertex firstVertex in _vertices)
            {
```

```
                var random = new Random();
                int randomNeighboursCount = random
                    .Next(0, GraphValidator.MaximalVertexDegree -
firstVertex.Degree + 1);

                var otherVertices = _vertices.Except(new List<GraphVertex>() {
firstVertex })
                    .Where(v => v.Degree !=
GraphValidator.MaximalVertexDegree).ToList();

                for (var i = 0; (i < randomNeighboursCount) && (i <
GetVerticesCount() - 1); i++)
                {
                    GraphVertex secondVertex;

                    do
                    {
                        var randomVerticeIndex = random.Next(otherVertices.Count);
                        secondVertex = otherVertices[randomVerticeIndex];
                    } while (EdgeExists(firstVertex.Number, secondVertex.Number));

                    AddEdge(firstVertex.Number, secondVertex.Number);
                }
            }
        }

        private void GenerateMinimalGraphEdgesCovering()
        {
            foreach (GraphVertex firstVertex in _vertices)
            {
                var otherVertices = _vertices.Where(vertex => vertex.Number !=
firstVertex.Number).ToList();
                var secondVertex = otherVertices
                    .Find(v => (v.Degree != GraphValidator.MaximalVertexDegree)
                    && (v.Degree == otherVertices.Min(v => v.Degree)));

                AddEdge(firstVertex.Number, secondVertex.Number);
            }
        }

        public bool EdgeExists(int firstVertexNumber, int secondVertexNumber)
        {
            return _edges.Contains(FindEdgeByEndsNumbers(firstVertexNumber,
secondVertexNumber));
        }

        public GraphEdge FindEdgeByEndsNumbers(int firstVertexNumber, int
secondVertexNumber)
        {
            return _edges
```

```
                .Find(e => ((e.FirstVertex.Number == firstVertexNumber)
                && (e.SecondVertex.Number == secondVertexNumber))
                || ((e.SecondVertex.Number == firstVertexNumber)
                && (e.FirstVertex.Number == secondVertexNumber)));
        }

        public int GetVerticesCount()
        {
            return _vertices.Count;
        }

        /// <exception cref="ArgumentException"></exception>
        /// <exception cref="ArgumentOutOfRangeException"></exception>
        public void AddEdge(int firstVertexNumber, int secondVertexNumber)
        {
            var firstVertex = FindVertexByNumber(firstVertexNumber);
            var secondVertex = FindVertexByNumber(secondVertexNumber);

            if (firstVertex is null)
            {
                throw new ArgumentException
                    ("A vertice with specified number doesn't exist",
nameof(firstVertexNumber));
            }

            if (secondVertex is null)
            {
                throw new ArgumentException
                    ("A vertice with specified number doesn't exist",
nameof(secondVertexNumber));
            }

            _edges.Add(new GraphEdge(firstVertex, secondVertex));
        }

        public GraphVertex FindVertexByNumber(int vertexNumber)
        {
            return _vertices.Find(v => v.Number == vertexNumber);
        }

        public void AddVertice()
        {
            _vertices.Add(new GraphVertex(_nextVertexNumber++));
        }

        public bool ContainsVertexNumber(int vertexNumber)
        {
            return _vertices.Exists(v => v.Number == vertexNumber);
        }
```

```
        public override string ToString()
        {
            var builder = new StringBuilder();

            foreach (GraphEdge edge in _edges)
            {
                builder.Append($" {edge} |");
            }

            return builder.ToString();
        }

        public IEnumerator<GraphEdge> GetEnumerator()
        {
            foreach (GraphEdge edge in _edges)
            {
                yield return edge;
            }
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }
}
```

```
using System.Text;

namespace ShortestPathProblemLogic
{
    public class Chromosome : ICloneable
    {
        private List<int> _ownedVerticesNumbers = new List<int>();

        public void ReplaceVertexNumber(int index, int numberToReplaceBy)
        {
            _ownedVerticesNumbers[index] = numberToReplaceBy;
        }

        /// <exception cref="ArgumentOutOfRangeException"></exception>
        public void InsertVertexNumberAt(int index, int vertexNumber, GraphOfSites
graph)
        {
            if (!graph.ContainsVertexNumber(vertexNumber))
            {
                throw new ArgumentOutOfRangeException(nameof(vertexNumber));
            }

            _ownedVerticesNumbers.Insert(index, vertexNumber);
```

```
        }

        public List<int> GetVerticesNumbersExceptEnds()
        {
            return _ownedVerticesNumbers
                .Except(new List<int> { _ownedVerticesNumbers.First(),
_ownedVerticesNumbers.Last() }).ToList();
        }

        public int GetLength(GraphOfSites graph)
        {
            return graph.GetEdgesWhichConnectVertices(GetVerticesNumbers())
                .Sum(e => e.Length);
        }

        public int FirstVertexNumber()
        {
            return _ownedVerticesNumbers.FirstOrDefault();
        }

        public void AddVerticesNumbersRange(IEnumerable<int> verticesNumbers)
        {
            _ownedVerticesNumbers.AddRange(verticesNumbers);
        }

        public List<int> GetVerticesNumbers()
        {
            return ((int[])_ownedVerticesNumbers.ToArray().Clone()).ToList();
        }

        public void RemoveVerticesNumbersExceptStart()
        {
            _ownedVerticesNumbers.RemoveRange(1, _ownedVerticesNumbers.Count - 1);
        }

        public void RemoveVertexNumber(int vertexNumber)
        {
            _ownedVerticesNumbers.Remove(vertexNumber);
        }

        public int LastVertexNumber()
        {
            return _ownedVerticesNumbers.LastOrDefault();
        }

        /// <exception cref="ArgumentException"></exception>
        public void AddVertexNumber(int vertexNumber)
        {
            if (ContainsVertexNumber(vertexNumber))
            {
```

```
                throw new ArgumentException("Specified vertex number already exists
in the list");
            }

            _ownedVerticesNumbers.Add(vertexNumber);
        }

        public bool ContainsVertexNumber(int vertexNumber)
        {
            return _ownedVerticesNumbers.Contains(vertexNumber);
        }

        public override string ToString()
        {
            var builder = new StringBuilder();
            _ownedVerticesNumbers.ForEach(n => builder.Append($" {n} "));

            return builder.ToString();
        }

        public object Clone()
        {
            var result = new Chromosome();

            result.AddVerticesNumbersRange(GetVerticesNumbers());

            return result;
        }
    }
}
```

```
using ShortestPathProblemLogic.Crossover;
using ShortestPathProblemLogic.Mutation;
using ShortestPathProblemLogic.LocalImprovement;

namespace ShortestPathProblemLogic
{
    public class GeneticProblemSolver
    {
        private GraphOfSites _graph;
        private PopulationGenerator _populationGenerator;

        public ICrossoverable Crossover { get; set; }

        public IMutationMakable MutationMaker { get; set; }

        public ILocalImprovable LocalImprover { get; set; }

        public GeneticProblemSolver(GraphOfSites graph, PopulationGenerator
populationGenerator,
```

```
            ICrossoverable crossover, IMutationMakable mutationMaker,
ILocalImprovable localImprover)
        {
            _graph = graph;
            _populationGenerator = populationGenerator;
            Crossover = crossover;
            MutationMaker = mutationMaker;
            LocalImprover = localImprover;
        }

        public List<GraphEdge> Solve(int chromosomesCount, int iterationsCount)
        {
            _populationGenerator.GenerateChromosomes(chromosomesCount);
            var maximalMutationProbabilityLevel = 0.7;

            FileHandler.WriteLineToFile(DateTime.Now.ToString());

            for (var i = 0; i < iterationsCount; i++)
            {
                if ((i + 1) % 10 == 0)
                {
                    foreach (Chromosome chromosome in _populationGenerator)
                    {
                        FileHandler.WriteLineToFile($"{chromosome}\n");
                    }

                    FileHandler.WriteLineToFile($"The shortest path:
{GetShortestPathLength()}\n");

                    foreach (GraphEdge edge in GetShortestPath())
                    {
                        FileHandler.WriteLineToFile($" {edge} |");
                    }
                }

                var offspring = Crossover.MakeCrossover();

                if (offspring is null)
                {
                    continue;
                }

                var random = new Random();
                var mutationProbability = random.NextDouble();

                if (mutationProbability <= maximalMutationProbabilityLevel)
                {
                    MutationMaker.MakeMutation(offspring);
                }
```

```
                LocalImprover.MakeImprovement(offspring);

                _populationGenerator.AddChromosome(offspring);
                _populationGenerator.RemoveChromosome(GetLongestChromosome());
            }

            return GetShortestPath();
        }

        public Chromosome GetLongestChromosome()
        {
            return _populationGenerator.MaxBy(c => c.GetLength(_graph));
        }

        public int GetShortestPathLength()
        {
            return _populationGenerator.Min(c => c.GetLength(_graph));
        }

        public List<GraphEdge> GetShortestPath()
        {
            return _graph.GetEdgesWhichConnectVertices(_populationGenerator
                .MinBy(c => c.GetLength(_graph)).GetVerticesNumbers());
        }
    }
}
```

```
namespace ShortestPathProblemLogic.Mutation
{
    public interface IMutationMakable
    {
        public void MakeMutation(Chromosome offspring);
    }
}
```

```
namespace ShortestPathProblemLogic.Mutation
{
    public class InsertionMutationMaker : IMutationMakable
    {
        private GraphOfSites _graph;

        public InsertionMutationMaker(GraphOfSites graph)
        {
            _graph = graph;
        }

        public void MakeMutation(Chromosome offspringToMutate)
        {
            int indexToInsertVertexNumberAt;
```

```
            int numberToInsert = 0;
            var verticesNumbers = offspringToMutate.GetVerticesNumbers();
            var triedIndices = new List<int>();

            do
            {
                var random = new Random();
                indexToInsertVertexNumberAt = random.Next(1,
verticesNumbers.Count);
                triedIndices.Add(indexToInsertVertexNumberAt);

                var leftVertexNumber = verticesNumbers[indexToInsertVertexNumberAt
- 1];
                var leftVerticesNumbers = _graph
                    .GetOtherVerticesOfEdgesWithSpecifiedVertexNumber(leftVertexNum
ber);

                var rightVertexNumber =
verticesNumbers[indexToInsertVertexNumberAt];
                var rightVerticesNumbers = _graph
                    .GetOtherVerticesOfEdgesWithSpecifiedVertexNumber(rightVertexNu
mber);

                var commonVerticesNumbers =
leftVerticesNumbers.Intersect(rightVerticesNumbers)
                    .Except(verticesNumbers).ToList();

                if (commonVerticesNumbers.Count != 0)
                {
                    var randomIndexOfCommonVertexNumber =
random.Next(commonVerticesNumbers.Count);
                    numberToInsert =
commonVerticesNumbers[randomIndexOfCommonVertexNumber];
                }
            } while ((numberToInsert == 0) && (triedIndices.Count !=
verticesNumbers.Count - 1));

            if (numberToInsert != 0)
            {
                offspringToMutate.InsertVertexNumberAt(indexToInsertVertexNumberAt,
numberToInsert, _graph);
            }
        }
    }
}
```

```
namespace ShortestPathProblemLogic.Mutation
{
    public class InversionMutationMaker : IMutationMakable
    {
```

```
        private GraphOfSites _graph;

        public InversionMutationMaker(GraphOfSites graph)
        {
            _graph = graph;
        }

        public void MakeMutation(Chromosome offspring)
        {
            var (firstVertexNumberPosition, secondVertexNumberPosition) =
GetBorderVerticesPositions
                (offspring.GetVerticesNumbers());

            MakeVerticesSubsetInversion
                (offspring.GetVerticesNumbers(), firstVertexNumberPosition,
secondVertexNumberPosition);
        }

        /// <exception cref="ArgumentOutOfRangeException"></exception>
        private void MakeVerticesSubsetInversion(List<int> verticesNumbers, int
firstBorderVertexNumberPosition,
            int secondBorderVertexNumberPosition)
        {
            bool firstPositionIsLessThanSecond = (firstBorderVertexNumberPosition <
secondBorderVertexNumberPosition);
            int leftIndex;
            int rightIndex;

            if (firstPositionIsLessThanSecond)
            {
                leftIndex = firstBorderVertexNumberPosition;
                rightIndex = secondBorderVertexNumberPosition;
            }
            else
            {
                leftIndex = secondBorderVertexNumberPosition;
                rightIndex = firstBorderVertexNumberPosition;
            }

            var leftBorderNumber = verticesNumbers[leftIndex];
            var rightBorderNumber = verticesNumbers[rightIndex];

            var firstEdgeToUpdate = _graph
                .FindEdgeByEndsNumbers(leftBorderNumber, verticesNumbers[leftIndex
+ 1]);
            var secondEdgeToUpdate = _graph
                .FindEdgeByEndsNumbers(rightBorderNumber,
verticesNumbers[rightIndex - 1]);

            verticesNumbers.Reverse(leftIndex + 1, rightIndex - leftIndex - 1);
```

```
            _graph.UpdateEdgeVertexNumber(firstEdgeToUpdate, leftBorderNumber,
verticesNumbers[leftIndex + 1]);
            _graph.UpdateEdgeVertexNumber(secondEdgeToUpdate, rightBorderNumber,
verticesNumbers[rightIndex - 1]);
        }

        /// <exception cref="ArgumentException"></exception>
        private (int, int) GetBorderVerticesPositions(List<int>
verticesNumbersToCreateSubsetIn)
        {
            int minimalVerticesNumbersCountToGetPositions = 4;

            if (verticesNumbersToCreateSubsetIn.Count <
minimalVerticesNumbersCountToGetPositions)
            {
                throw new ArgumentException("The count of vertices numbers mustn't
be less than 3");
            }

            var random = new Random();

            var firstPosition = random.Next(verticesNumbersToCreateSubsetIn.Count);
            int secondPosition;

            do
            {
                secondPosition =
random.Next(verticesNumbersToCreateSubsetIn.Count);
            } while (Math.Abs(firstPosition - secondPosition) < 3);

            return (firstPosition, secondPosition);
        }
    }
}
```

```
namespace ShortestPathProblemLogic.LocalImprovement
{
    public interface ILocalImprovable
    {
        public void MakeImprovement(Chromosome offspring);
    }
}
```

```
namespace ShortestPathProblemLogic.LocalImprovement
{
    public class ProfitableVertexReplacementLocalImprover : ILocalImprovable
    {
        private GraphOfSites _graph;
```

```
        public ProfitableVertexReplacementLocalImprover(GraphOfSites graph)
        {
            _graph = graph;
        }

        public void MakeImprovement(Chromosome offspringToImprove)
        {
            int indexOfVertexNumberToReplace;
            var verticesNumbers = offspringToImprove.GetVerticesNumbers();
            var triedIndices = new List<int>();
            var replacementIsDone = false;

            do
            {
                var random = new Random();
                indexOfVertexNumberToReplace = random.Next(1, verticesNumbers.Count
- 1);

                if (!triedIndices.Contains(indexOfVertexNumberToReplace))
                {
                    triedIndices.Add(indexOfVertexNumberToReplace);
                }

                var leftVertexNumber = verticesNumbers[indexOfVertexNumberToReplace
- 1];
                var leftVerticesNumbers = _graph
                    .GetOtherVerticesOfEdgesWithSpecifiedVertexNumber(leftVertexNum
ber);

                var rightVertexNumber =
verticesNumbers[indexOfVertexNumberToReplace + 1];
                var rightVerticesNumbers = _graph
                    .GetOtherVerticesOfEdgesWithSpecifiedVertexNumber(rightVertexNu
mber);

                var commonVerticesNumbers =
leftVerticesNumbers.Intersect(rightVerticesNumbers)
                    .Except(verticesNumbers).ToList();

                if (commonVerticesNumbers.Count != 0)
                {
                    var currentPathLength = offspringToImprove.GetLength(_graph);

                    foreach (int vertexNumber in commonVerticesNumbers)
                    {
                        var offstringClone =
(Chromosome)offspringToImprove.Clone();

                        offstringClone.ReplaceVertexNumber(indexOfVertexNumberToRep
lace, vertexNumber);
```

```
                        if ((offstringClone.GetLength(_graph) < currentPathLength))
                        {
                            offspringToImprove.ReplaceVertexNumber(indexOfVertexNum
berToReplace,
                                vertexNumber);
                            currentPathLength =
offspringToImprove.GetLength(_graph);
                            replacementIsDone = true;
                        }
                    }
                }
            } while (!replacementIsDone && (triedIndices.Count !=
verticesNumbers.Count - 2));
        }
    }
}
```

```
namespace ShortestPathProblemLogic.Crossover
{
    public interface ICrossoverable
    {
        public Chromosome MakeCrossover();
    }
}
```

```
namespace ShortestPathProblemLogic.Crossover
{
    public class SinglePointCrossover : ICrossoverable
    {
        private PopulationGenerator _populationGenerator;
        private GraphOfSites _graph;

        public SinglePointCrossover(PopulationGenerator populationGenerator,
GraphOfSites graph)
        {
            _populationGenerator = populationGenerator;
            _graph = graph;
        }

        private int FindCommonVertexNumber(Chromosome firstParent, Chromosome
secondParent)
        {
            var firstParentVerticesToCompare =
firstParent.GetVerticesNumbersExceptEnds();
            var secondParentVerticesToCompare =
secondParent.GetVerticesNumbersExceptEnds();
```

```
            return
firstParentVerticesToCompare.Intersect(secondParentVerticesToCompare).FirstOrDefaul
t();
        }

        private Chromosome GetCrossoverableRandomParent(Chromosome anotherParent)
        {
            Chromosome result = null;
            var chromosomesExceptFirstParent = _populationGenerator
                .Except(new List<Chromosome> { anotherParent }).ToList();

            var random = new Random();
            var randomChromosomeIndex =
random.Next(chromosomesExceptFirstParent.Count);
            result = chromosomesExceptFirstParent[randomChromosomeIndex];

            return result;
        }

        public Chromosome MakeCrossover()
        {
            var result = new Chromosome();

            var firstParent = GetCrossoverableRandomParent(null);
            var secondParent = GetCrossoverableRandomParent(firstParent);
            var commonParentsVertexNumber = FindCommonVertexNumber(firstParent,
secondParent);

            if (commonParentsVertexNumber == 0)
            {
                return null;
            }

            var firstParentLeftPart = GetFirstParentLeftPart(firstParent,
commonParentsVertexNumber);
            result.AddVerticesNumbersRange(firstParentLeftPart);

            result.AddVertexNumber(commonParentsVertexNumber);

            var secondParentRightPart = GetSecondParentRightPart(secondParent,
commonParentsVertexNumber);
            result.AddVerticesNumbersRange(secondParentRightPart);

            return result;
        }

        private List<int> GetSecondParentRightPart(Chromosome secondParent, int
commonParentsVertexNumber)
        {
            var secondParentVerticesNumbers = secondParent.GetVerticesNumbers();
```

```
            var secondParentCommonVertexIndex =
secondParentVerticesNumbers.IndexOf(commonParentsVertexNumber);

            return secondParentVerticesNumbers
                .GetRange(secondParentCommonVertexIndex + 1,
                secondParentVerticesNumbers.Count - secondParentCommonVertexIndex -
1);
        }

        private List<int> GetFirstParentLeftPart(Chromosome firstParent, int
commonParentsVertexNumber)
        {
            return firstParent.GetVerticesNumbers()
                .TakeWhile(num => num != commonParentsVertexNumber).ToList();
        }
    }
}
```

```
namespace ShortestPathProblemLogic.Crossover
{
    public class UniformCrossover : ICrossoverable
    {
        private PopulationGenerator _populationGenerator;
        private GraphOfSites _graph;

        public UniformCrossover(PopulationGenerator populationGenerator,
GraphOfSites graph)
        {
            _populationGenerator = populationGenerator;
            _graph = graph;
        }

        public Chromosome MakeCrossover()
        {
            var result = new Chromosome();

            var firstParent = GetCrossoverableRandomParent(null);
            var secondParent = GetCrossoverableRandomParent(firstParent);
            result.AddVertexNumber(firstParent.FirstVertexNumber());

            var firstParentVerticesNumbers =
firstParent.GetVerticesNumbersExceptEnds();
            var secondParentVerticesNumbers =
secondParent.GetVerticesNumbersExceptEnds();

            var isFirstParentBiggerThanSecond = firstParentVerticesNumbers.Count
                > secondParentVerticesNumbers.Count;

            List<int> tailOfBiggerParentVerticesNumbers = null;
            int lessParentVerticesNumbersCount;
```

```
            if (isFirstParentBiggerThanSecond)
            {
                lessParentVerticesNumbersCount = secondParentVerticesNumbers.Count;
                tailOfBiggerParentVerticesNumbers =
                    firstParentVerticesNumbers.Skip(secondParentVerticesNumbers.Cou
nt).ToList();
            }
            else
            {
                lessParentVerticesNumbersCount = firstParentVerticesNumbers.Count;
                tailOfBiggerParentVerticesNumbers =
                    secondParentVerticesNumbers.Skip(firstParentVerticesNumbers.Cou
nt).ToList();
            }

            for (var i = 0; i < lessParentVerticesNumbersCount; i++)
            {
                var random = new Random();
                var indicatorOnWhichVertexNumberGoesToOffspring = random.Next(2);

                if (indicatorOnWhichVertexNumberGoesToOffspring == 0)
                {
                    if
(!result.ContainsVertexNumber(firstParentVerticesNumbers[i]))
                    {
                        result.AddVertexNumber(firstParentVerticesNumbers[i]);
                    }
                }
                else
                {
                    if
(!result.ContainsVertexNumber(secondParentVerticesNumbers[i]))
                    {
                        result.AddVertexNumber(secondParentVerticesNumbers[i]);
                    }
                }
            }

            result.AddVerticesNumbersRange(tailOfBiggerParentVerticesNumbers
                .Where(num => !result.ContainsVertexNumber(num)));
            result.AddVertexNumber(firstParent.LastVertexNumber());

            if
(_graph.GetEdgesWhichConnectVertices(result.GetVerticesNumbers()).Contains(null))
            {
                return null;
            }

            return result;
```

```
        }

        private int FindCommonVertexNumber(Chromosome firstParent, Chromosome
secondParent)
        {
            var firstParentVerticesToCompare =
firstParent.GetVerticesNumbersExceptEnds();
            var secondParentVerticesToCompare =
secondParent.GetVerticesNumbersExceptEnds();

            return
firstParentVerticesToCompare.Intersect(secondParentVerticesToCompare).FirstOrDefaul
t();
        }

        private Chromosome GetCrossoverableRandomParent(Chromosome anotherParent)
        {
            Chromosome result = null;
            var chromosomesExceptFirstParent = _populationGenerator
                .Except(new List<Chromosome> { anotherParent }).ToList();

            var random = new Random();
            var randomChromosomeIndex =
random.Next(chromosomesExceptFirstParent.Count);
            result = chromosomesExceptFirstParent[randomChromosomeIndex];

            return result;
        }
    }
}
using ShortestPathProblemLogic;
using ShortestPathProblemLogic.Crossover;
using ShortestPathProblemLogic.Mutation;
using ShortestPathProblemLogic.LocalImprovement;

namespace ShortestPathProblemRunner
{
    public class Program
    {
        public static void Main(string[] args)
        {
            FileHandler.ClearFile();

            System.Console.WriteLine("Operators of the first solving case:\n"
                + " - single point crossover\n"
                + " - insertion mutation\n"
                + " - vertex replacement local improvement\n");

            var firstCaseGraph = DataCatcher.CatchGraph();
```

```
            var firstCasePopulationGenerator =
DataCatcher.CatchPopulationGenerator(firstCaseGraph);
            ICrossoverable firstCaseCrossover =
                new SinglePointCrossover(firstCasePopulationGenerator,
firstCaseGraph);
            IMutationMakable mutationMaker = new
InsertionMutationMaker(firstCaseGraph);

            SolvingCases.UseSolvingCase(firstCaseGraph,
firstCasePopulationGenerator, firstCaseCrossover,
                mutationMaker);

            PrintHyphenLine();
            FileHandler.WriteLineToFile($"{new string('-', 80)}\n");

            System.Console.WriteLine("Operators of the second solving case:\n"
                + " - uniform crossover\n"
                + " - insertion mutation\n"
                + " - vertex replacement local improvement\n");

            var secondCaseGraph = DataCatcher.CatchGraph();
            var secondCasePopulationGenerator =
DataCatcher.CatchPopulationGenerator(secondCaseGraph);
            ICrossoverable secondCaseCrossover = new
UniformCrossover(secondCasePopulationGenerator, secondCaseGraph);
            IMutationMakable secondCaseMutationMaker = new
InsertionMutationMaker(secondCaseGraph);

            SolvingCases.UseSolvingCase(secondCaseGraph,
secondCasePopulationGenerator, secondCaseCrossover,
                secondCaseMutationMaker);
        }

        public static void PrintHyphenLine()
        {
            System.Console.WriteLine(new string('-', 70));
        }
    }
}
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

```
$ dotnet run
Operators of the first solving case:
 - single point crossover
 - insertion mutation
 - vertex replacement local improvement

Enter initial vertices' count (required 300): 300
Enter start vertex number: 1
Enter end vertex number: 200
Enter the chromosomes count: 50
Enter iterations count: 1000
Please, wait. Solving the shortest path problem...

Shortest path: 183
 261 <--47--> 1 |
 261 <--78--> 284 |
 18 <--37--> 284 |
 18 <--21--> 200 |
------------------------------------------------------------------------
Operators of the second solving case:
 - uniform crossover
 - insertion mutation
 - vertex replacement local improvement

Enter initial vertices' count (required 300): 300
Enter start vertex number: 1
Enter end vertex number: 11
Enter the chromosomes count: 50
Enter iterations count: 1000
Please, wait. Solving the shortest path problem...

Shortest path: 220
 1 <--98--> 74 |
 74 <--122--> 11 |
```

Рисунок 3.1 – результат роботи програми для одноточкового та рівномірного операторів схрещування

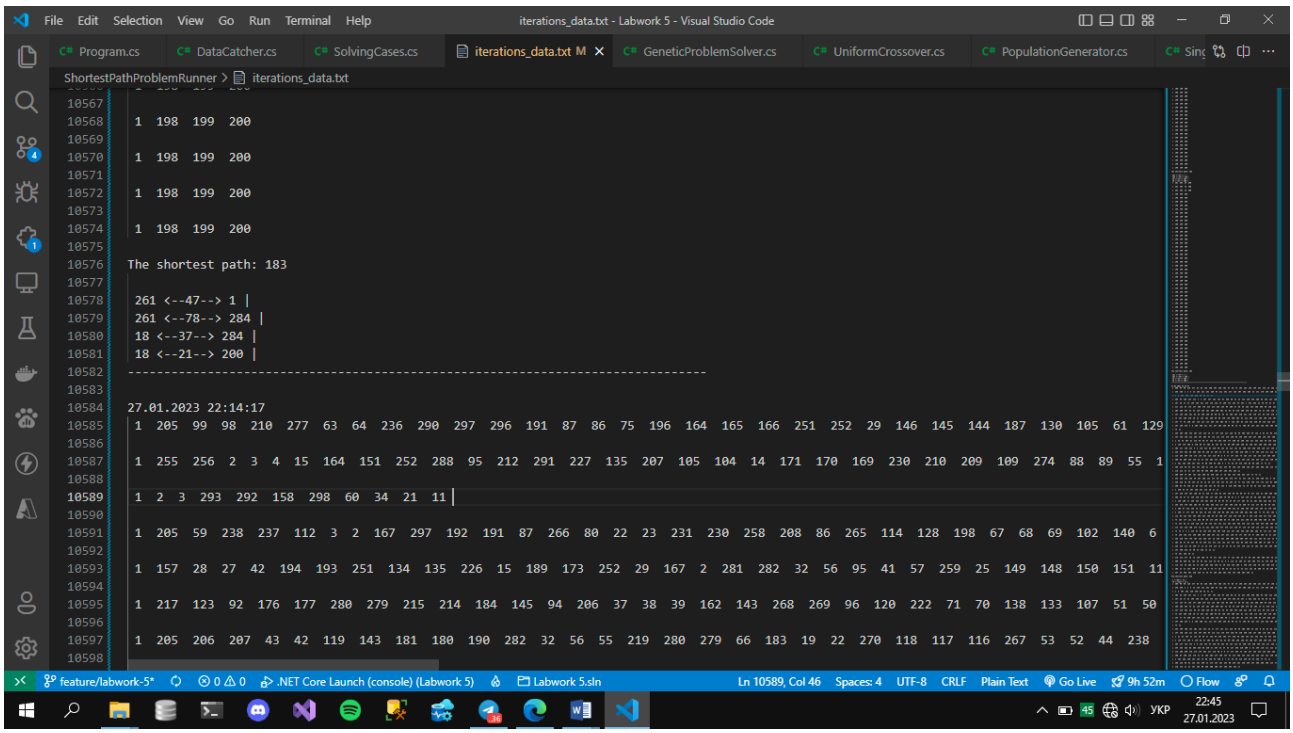


Рисунок 3.2 – запис до текстового файлу значень хромосом та найкоротшого шляху разом із його представленням у вигляді сукупності дуг кожні 10 ітерацій програми

3.3 Тестування алгоритму

Перед переходом до суті, доречно буде описати існуючі обмеження, які визначаються умовою задачі. Тобто, в результаті я маю граф на 300 вершин, степені яких коливаються від 1 до 10 включно, а відстані між ними від 5 до 150 включно. Також, окрім цього, є ще параметри, які впливають на поведінку алгоритму.

3.3.1 Дослідження параметру мутації

Мутація в генетичному алгоритмі несе за собою невизначеність та сприяє захопленню нових вершин графу в околицях поточних рішень. Вона сприяє розростанню популяції та її бажанню не сидіти на місці.

Таблиця 3.1 – значення цільової функції зі зміною параметра мутації

Ймовірність мутації	Номер дослідження	Значення цільової функції
0	1	164
	2	208
	3	145
	4	190
	5	139
	Середнє	169
0,2	1	179
	2	222
	3	217
	4	220
	5	175
	Середнє	202

Продовження таблиці 3.1

Ймовірність мутації	Номер дослідження	Значення цільової функції
0,4	1	280
	2	373
	3	234
	4	484
	5	177
	Середнє	309
0,6	1	227
	2	141
	3	204
	4	158
	5	161
	Середнє	178
0,8	1	176
	2	183
	3	180
	4	315
	5	76
	Середнє	186
1	1	143
	2	185
	3	227
	4	234
	5	190
	Середнє	195

Рисунок 3.3 – залежність значень цільової функції від імовірності мутації

ВИСНОВОК

В рамках даної лабораторної роботи я застосував оптимальне рішення проблеми найкоротшого шляху на практиці за допомогою генетичного алгоритму.

Це сталося завдяки тому, що я підбирав прийнятні параметри для вирішення конкретної проблеми. Я створив оператори схрещування, мутації та локального покращення, кожен вид яких діяв по-своєму, направлений на конкретну область застосування алгоритму.

Наприклад, мутація, котра спрацьовує над нащадком в залежності від завчасно нарахованої ймовірності, ефективна не завжди. Як можна помітити, я створив графік, котрий відображає залежність між значеннями цільової функції та ймовірністю мутації. Проте, її застосування стає доцільним лише за малих або великих шансів спрацювання.

Під час виконання завдань, мною було прийняте рішення записувати дані, отримані під час роботи програми, до файлу, котрий розташовується у проєкті, що запускає програму.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 11.12.2022 включно максимальний бал дорівнює – 5. Після 11.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- покроковий алгоритм – 15%;
- програмна реалізація алгоритму – 50%;
- тестування алгоритму– 30%;
- висновок – 5%.