

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)

ІІІ-13 Бондаренко М.В.
(шифр, прізвище, ім'я, по батькові)

Перевірив

Сопов О.О.
(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	12
3.2.1	<i>Вихідний код.....</i>	<i>12</i>
3.2.2	<i>Приклади роботи</i>	<i>17</i>
3.3	ТЕСТУВАННЯ АЛГОРИТМУ	18
	ВИСНОВОК	21
	КРИТЕРІЇ ОЦІНЮВАННЯ	22

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 ступенем 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.

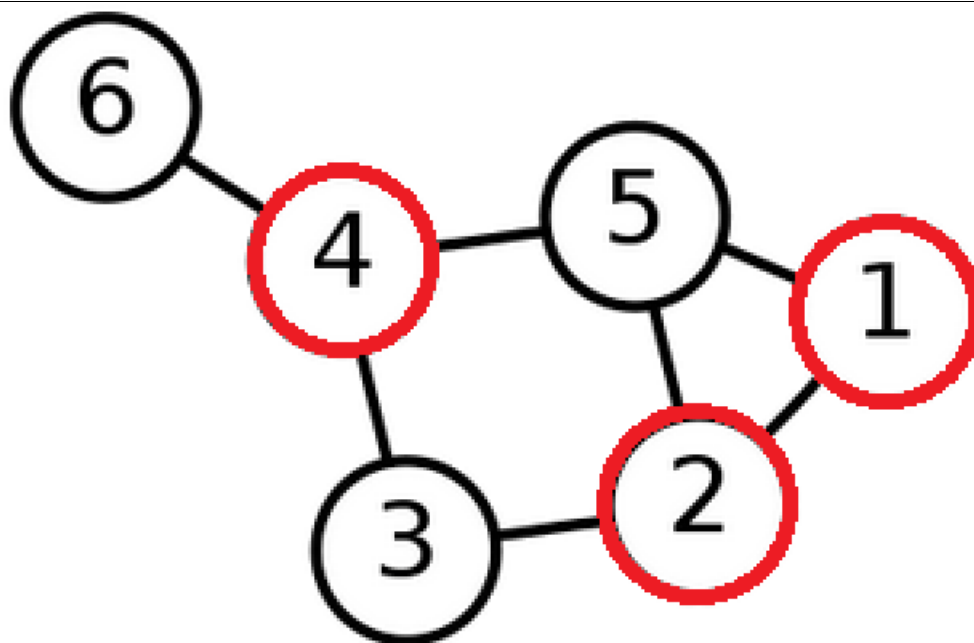
Зробити узагальнений висновок в якому обов'язково описати залежність якості розв'язку від вхідних параметрів.

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

	<p>сумарна вага не перевищувала задану, а сумарна цінність була максимальною.</p> <p>Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика.</p>
2	<p>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.</p> <p>Розглядається симетричний, асиметричний та змішаний варіанти.</p> <p>В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напрямку, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра.</p> <p>У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.</p> <p>У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напрямку руху.</p> <p>Застосування:</p> <ul style="list-style-type: none"> — доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); — доставка води;

	<ul style="list-style-type: none"> – моніторинг об'єктів; – поповнення банкоматів готівкою; – збір співробітників для доставки вахтовим методом.
3	<p>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.</p> <p>Застосування:</p> <ul style="list-style-type: none"> – розкладу для освітніх установ; – розкладу в спорті; – планування зустрічей, зборів, інтерв'ю; – розклади транспорту, в тому числі - авіатранспорту; – розкладу для комунальних служб;
4	<p>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.</p> <p>Задача вершинного покриття полягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).</p> <p>На вході: Граф $G = (V, E)$.</p> <p>Результат: множина $C \subseteq V$ - найменше вершинне покриття графа G.</p>



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

	<p>не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.</p> <p>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.</p>
--	--

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

№	Алгоритми і досліджувані параметри
1	<p>Генетичний алгоритм:</p> <ul style="list-style-type: none"> - оператор схрещування (мінімум 3); - мутація (мінімум 2); - оператор локального покращення (мінімум 2).
2	<p>Мурашиний алгоритм:</p> <ul style="list-style-type: none"> – α; – β; – ρ; – L_{min}; – кількість мурах M і їх типи (елітні, тощо...); – маршрути з однієї чи різних вершин.
3	<p>Бджолиний алгоритм:</p> <ul style="list-style-type: none"> – 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

3.1 Покроковий алгоритм

1. Ініціалізуйте об'єкт «Problem» і масив «CurrentGeneration», передавши об'єкт проблеми конструктору «Algorithm» і викликавши метод «GetInitialPopulation» із класу «Util».
2. Метод «GetBestSolution» повертає рішення з найменшою вартістю з масиву «CurrentGeneration».
3. Метод "EvolveGeneration" виконує одну ітерацію генетичного алгоритму.
4. Він створює пул розведення найкращих рішень із масиву «CurrentGeneration», беручи найкращі рішення «ElitePoolSize», відсортовані за їх вартістю.
5. Він також вибирає найкращі рішення "ElitePoolSize / 5" із племінного пулу та зберігає його в найкращому масиві.
6. Він розраховує ймовірності вибору кожного рішення в пулі розведення за допомогою методу "GetProbabilities" з класу "Util".
7. Він створює нову популяцію рішень із найкращими рішеннями "ElitePoolSize / 5" із найкращого масиву та рішеннями, що залишилися, створеними шляхом розведення та мутації рішень із пулу розведення.
8. Метод «CreateOffspring» приймає два рішення як вхідні дані, витягує першу половину генів із першого рішення, об'єднує гени двох рішень і повертає нове рішення.
9. Метод «ExtractFirstHalfGenes» приймає рішення як вхідні дані та витягує першу половину генів із шляху рішення.
10. Метод «CombinePaths» приймає як вхідні дані два рішення та масив першої половини генів із першого рішення, об'єднує гени двох рішень і повертає новий шлях.
11. Метод "RunIterations" запускає метод "EvolveGeneration" для вказаної кількості ітерацій.
12. Клас «Algorithm» використовує ці методи, щоб розвивати популяцію протягом заданої кількості ітерацій і отримує найкраще рішення з популяції, яке повертає метод «GetBestSolution».

13. Метод "EvolveGeneration()" викликається в циклі для певної кількості ітерацій, і найкраще рішення з кожної ітерації повертається методом "GetBestSolution()".
14. Алгоритм продовжує розвивати популяцію, доки не досягне критерію зупинки (наприклад, певної кількості ітерацій або бажаного значення функції вартості), визначеного користувачем.
15. Після виконання критерію зупинки алгоритм повертає найкраще рішення, знайдене під час ітерацій.
16. Алгоритм використовує генетичний алгоритм із комбінацією операторів відбору, кросинговеру та мутації для еволюції популяції та обчислення ймовірностей відбору рішень із популяції.

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

Constant.cs

```
namespace lab5;

public abstract class Constant
{
    public const int MaxPopulation = 1000;
    public const int ElitePoolSize = 100;
    public const string DataFile = "problem.json";
    public const int TotalCities = 300;
    private const int BatchNumber = 6;
    public const int BatchSize = TotalCities / BatchNumber;
    public const double Probability = 0.3;
}
```

Program.cs

```
using System.Text;

namespace lab5
{
    class Program
    {
        public static void Main(string[] args)
        {
            var result = new StringBuilder();
            var problem = new Problem();
            var algorithm = new Algorithm(problem);
            for (int i = 0; i < 100; i++)
            {
                algorithm.RunIterations(100);
                var bestSolution = algorithm.GetBestSolution();
                var line = $"iteration count: {i * 100 + 100}, cost on this
iteration: {bestSolution.Cost}";
                result.AppendLine(line);
                Console.WriteLine(line);
            }
        }
    }
}
```

Solution.cs

```
namespace lab5;

public struct Solution
{
    public int[] Path { get; }
    private Problem Problem { get; }
    public int Cost { get; }

    public Solution(Problem problem, int[] path)
    {
        Path = path;
        Problem = problem;
        Cost = Problem.GetCost(path);
    }
}
```

Util.cs

```
using System.Security.Cryptography;

namespace lab5;

public class Util
{
    public static double[] GetProbabilities(Solution[] solutions)
    {
        var values = new double[solutions.Length];
        var sum = 0.0;
        for (int i = 0; i < solutions.Length; i++)
        {
            values[i] = 1.0 / solutions[i].Cost;
            sum += values[i];
        }

        for (int i = 0; i < values.Length; i++)
        {
            values[i] /= sum;
        }

        return values;
    }

    public static Solution[] GetInitialPopulation(Problem problem)
    {
        Random rand = new Random();
        var population = new Solution[Constant.MaxPopulation];
        var random = Enumerable.Range(0, Constant.TotalCities).OrderBy(_ =>
rand.Next()).ToArray();
        for (int i = 0; i < Constant.MaxPopulation; i++)
        {
            population[i] = new Solution(problem, random);
        }
        return population;
    }

    public static int ChooseSolution(IReadOnlyList<double> probabilities)
    {
        Random rand = new Random();
        var random = rand.NextDouble();
        var sum = 0.0;
        for (int i = 0; i < probabilities.Count; i++)
        {

```

```

        sum += probabilities[i];
        if (sum > random)
        {
            return i;
        }
    }

    return probabilities.Count - 1;
}

public static int[][] GenerateMatrix()
{
    int[][] matrix = new int[Constant.TotalCities][];
    for (int i = 0; i < Constant.TotalCities; i++)
    {
        matrix[i] = new int[Constant.TotalCities];
        for (int j = 0; j < Constant.TotalCities; j++)
        {
            if (j > i)
            {
                matrix[i][j] = RandomNumberGenerator.GetInt32(5, 150);
            }
            else if (j == i)
            {
                matrix[i][j] = int.MaxValue;
            }
            else
            {
                matrix[i][j] = matrix[j][i];
            }
        }
    }

    return matrix;
}

public static void Mutate(Solution solution)
{
    Random random = new Random();
    var first = random.Next(0, solution.Path.Length);
    var second = random.Next(0, solution.Path.Length);
    (solution.Path[first], solution.Path[second]) = (solution.Path[second],
solution.Path[first]);
}
}

```

Algorithm.cs

```

namespace lab5;

public class Algorithm
{
    private readonly Problem _problem;

    private Solution[] CurrentGeneration { get; set; }

    public Algorithm(Problem problem)
    {
        _problem = problem;
        CurrentGeneration = Util.GetInitialPopulation(problem);
    }

    public Solution GetBestSolution()
    {

```

```

        return CurrentGeneration.MinBy(s => s.Cost);
    }

    public void EvolveGeneration()
    {
        var breedingPool = CurrentGeneration.OrderBy(s =>
s.Cost).Take(Constant.ElitePoolSize).ToArray();
        var best = breedingPool.Take(Constant.ElitePoolSize / 5).ToArray();

        var probabilities = Util.GetProbabilities(breedingPool);

        var newPopulation = new Solution[Constant.TotalCities];

        for (int i = 0; i < Constant.ElitePoolSize / 5; i++)
        {
            newPopulation[i] = best[i];
        }

        Random random = new Random();
        for (int i = Constant.ElitePoolSize / 5; i < Constant.TotalCities; i++)
        {
            newPopulation[i] =
CreateOffspring(breedingPool[Util.ChooseSolution(probabilities)],
                breedingPool[Util.ChooseSolution(probabilities)]);
            if (random.NextDouble() < Constant.Probability)
            {
                Util.Mutate(newPopulation[i]);
            }
        }

        CurrentGeneration = newPopulation;
    }

    private Solution CreateOffspring(Solution first, Solution second)
    {
        var firstHalfGenes = ExtractFirstHalfGenes(first);
        var path = CombinePaths(first, second, firstHalfGenes);
        return new Solution(_problem, path);
    }

    private int[] ExtractFirstHalfGenes(Solution first)
    {
        var firstHalfGenes = new int[Constant.TotalCities / 2];
        for (int i = 0; i < Constant.TotalCities / 2; i++)
        {
            firstHalfGenes[i] = first.Path[Constant.BatchSize * 2 * (i /
Constant.BatchSize) + i % Constant.BatchSize];
        }
        return firstHalfGenes;
    }

    private int[] CombinePaths(Solution first, Solution second, int[]
firstHalfGenes)
    {
        var path = new int[Constant.TotalCities];
        var currentSecond = 0;
        for (int i = 0; i < Constant.TotalCities; i++)
        {
            if ((i / Constant.BatchSize) % 2 == 0)
            {
                path[i] = first.Path[i];
            }
            else
            {

```

```

        while (firstHalfGenes.Contains(second.Path[currentSecond]))
        {
            currentSecond++;
        }

        path[i] = second.Path[currentSecond++];
    }
}
return path;
}

public void RunIterations(int iterationCount)
{
    for (int j = 0; j < iterationCount; j++)
    {
        EvolveGeneration();
    }
}
}

```

Problem.cs

```

using System.Security.Cryptography;
using System.Text.Json;

namespace lab5;

public class Problem
{
    private int[][] Matrix { get; set; }

    public Problem()
    {
        InitializeMatrix();
    }

    public int GetCost(int[] path)
    {
        var solution = 0;
        for (int i = 0; i < Constant.TotalCities - 1; i++)
        {
            solution += GetDistance(path[i], path[i + 1]);
        }

        solution += GetDistance(path[Constant.TotalCities - 1], path[0]);

        return solution;
    }

    private int GetDistance(int source, int destination) =>
Matrix[source][destination];

    private void InitializeMatrix()
    {
        if (MatrixFileExists())
        {
            Matrix = ReadMatrixFromFile();
            return;
        }

        Matrix = GenerateMatrix();
        SaveMatrixToFile();
    }
}

```



```

private bool MatrixFileExists()
{
    return File.Exists(Constant.DataFile);
}

private int[][] ReadMatrixFromFile()
{
    using var fs = new FileStream(Constant.DataFile, FileMode.Open);
    return JsonSerializer.Deserialize<int[][]>(fs)!;
}

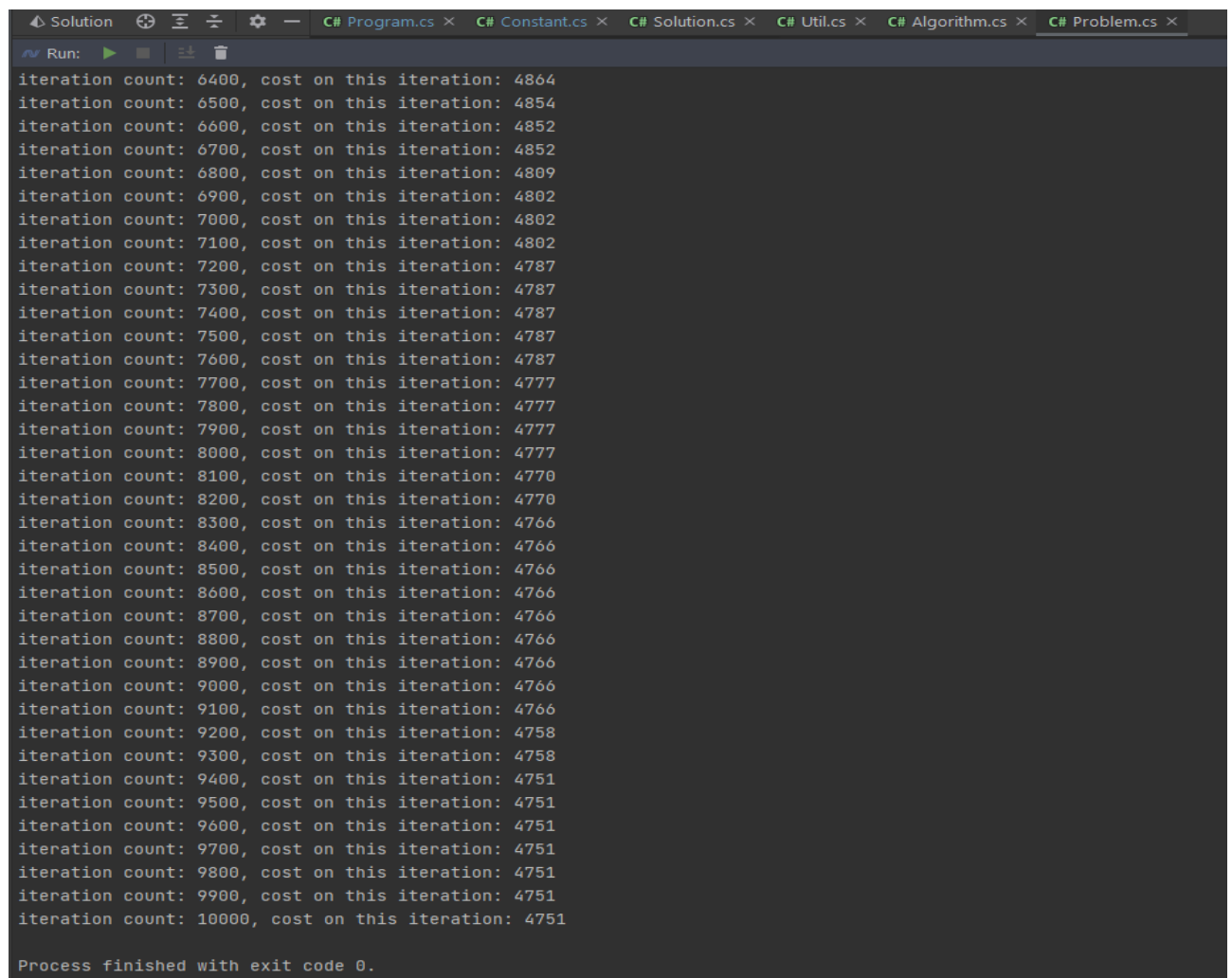
private int[][] GenerateMatrix()
{
    return Util.GenerateMatrix();
}

private void SaveMatrixToFile()
{
    using var fs = new FileStream(Constant.DataFile, FileMode.Create);
    JsonSerializer.Serialize(fs, Matrix);
}
}

```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.



```

iteration count: 6400, cost on this iteration: 4864
iteration count: 6500, cost on this iteration: 4854
iteration count: 6600, cost on this iteration: 4852
iteration count: 6700, cost on this iteration: 4852
iteration count: 6800, cost on this iteration: 4809
iteration count: 6900, cost on this iteration: 4802
iteration count: 7000, cost on this iteration: 4802
iteration count: 7100, cost on this iteration: 4802
iteration count: 7200, cost on this iteration: 4787
iteration count: 7300, cost on this iteration: 4787
iteration count: 7400, cost on this iteration: 4787
iteration count: 7500, cost on this iteration: 4787
iteration count: 7600, cost on this iteration: 4787
iteration count: 7700, cost on this iteration: 4777
iteration count: 7800, cost on this iteration: 4777
iteration count: 7900, cost on this iteration: 4777
iteration count: 8000, cost on this iteration: 4777
iteration count: 8100, cost on this iteration: 4770
iteration count: 8200, cost on this iteration: 4770
iteration count: 8300, cost on this iteration: 4766
iteration count: 8400, cost on this iteration: 4766
iteration count: 8500, cost on this iteration: 4766
iteration count: 8600, cost on this iteration: 4766
iteration count: 8700, cost on this iteration: 4766
iteration count: 8800, cost on this iteration: 4766
iteration count: 8900, cost on this iteration: 4766
iteration count: 9000, cost on this iteration: 4766
iteration count: 9100, cost on this iteration: 4766
iteration count: 9200, cost on this iteration: 4758
iteration count: 9300, cost on this iteration: 4758
iteration count: 9400, cost on this iteration: 4751
iteration count: 9500, cost on this iteration: 4751
iteration count: 9600, cost on this iteration: 4751
iteration count: 9700, cost on this iteration: 4751
iteration count: 9800, cost on this iteration: 4751
iteration count: 9900, cost on this iteration: 4751
iteration count: 10000, cost on this iteration: 4751

Process finished with exit code 0.

```

Рисунок 3.1 – виведення кількості ітерацій та вартість при них.

3.3 Тестування алгоритму

Протестуємо алгоритм на різних вхідних даних. Тестування буде проходити на 10000 ітераціях алгоритму з кроком в 100 ітерацій, буде отримано результат кожної ітерації, для схрещування буде використано 100 найкращих особин з покоління.

Вхідні дані: 10 сегментів, 0.1 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 5449
```

Вхідні дані: 6 сегментів, 0.1 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 5360
```

Вхідні дані: 20 сегментів, 0.1 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 5366
```

Вхідні дані: 6 сегментів, 0.2 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 5074
```

Вхідні дані: 6 сегментів, 0.3 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 4675
```

Вхідні дані: 6 сегментів, 0.4 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 4760
```

Вхідні дані: 10 сегментів, 0.3 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 5227
```

Вхідні дані: 4 сегменти, 0.3 ймовірність мутації

Результат:

```
iteration count: 10000, cost on this iteration: 4693
```

Отже, найкращі вхідні дані: 6 сегментів, 0.3 ймовірність, тепер проаналізуємо результати виконання програми за цих даних нижче.

За допомогою сервісу Desmos побудуємо графік залежності вартості від ітерацій

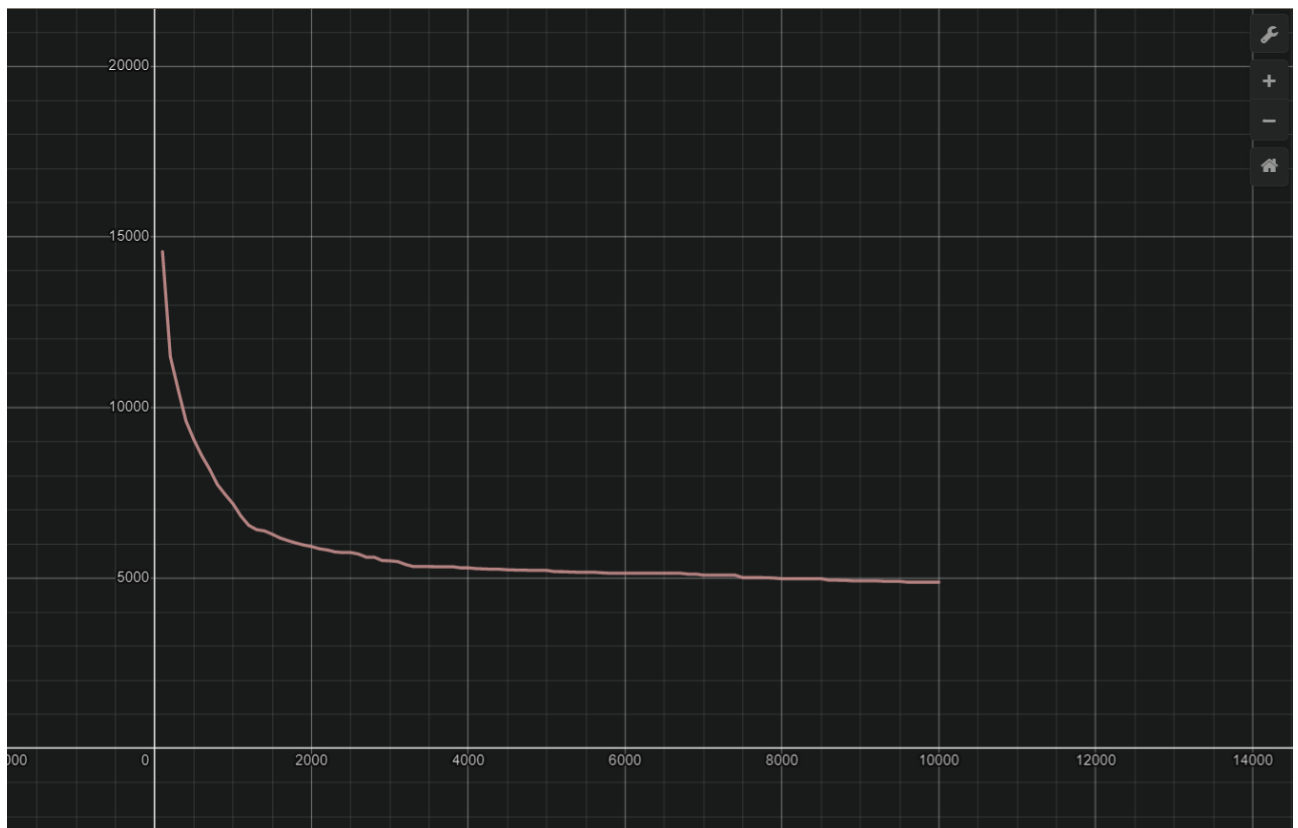


Рисунок 3.3 – графік залежності результату від ітерацій

Побудуємо таблицю з кількістю ітерацій та значеннями, що супроводжують їх

iter	cost	iter	cost	iter	cost	iter	cost	iter	cost
100	14568	2100	5860	4100	5278	6100	5144	8100	4982
200	11490	2200	5825	4200	5270	6200	5144	8200	4982

300	10525	2300	5768	4300	5260	6300	5144	8300	4982
400	9600	2400	5750	4400	5260	6400	5144	8400	4982
500	9058	2500	5750	4500	5244	6500	5144	8500	4982
600	8591	2600	5705	4600	5237	6600	5144	8600	4939
700	8196	2700	5612	4700	5237	6700	5144	8700	4939
800	7742	2800	5612	4800	5225	6800	5118	8800	4935
900	7449	2900	5516	4900	5225	6900	5118	8900	4921
1000	7175	3000	5506	5000	5225	7000	5087	9000	4921
1100	6819	3100	5484	5100	5191	7100	5087	9100	4921
1200	6546	3200	5399	5200	5189	7200	5087	9200	4921
1300	6421	3300	5337	5300	5180	7300	5087	9300	4908
1400	6383	3400	5337	5400	5173	7400	5087	9400	4905
1500	6286	3500	5337	5500	5173	7500	5015	9500	4905
1600	6175	3600	5334	5600	5173	7600	5015	9600	4882
1700	6098	3700	5334	5700	5156	7700	5015	9700	4882
1800	6029	3800	5334	5800	5144	7800	5013	9800	4882
1900	5970	3900	5301	5900	5144	7900	5002	9900	4882
2000	5926	4000	5301	6000	5144	8000	4982	10000	4882

Таблиця 3.1 – кількість ітерацій та значення при них.

ВИСНОВОК

В рамках даної лабораторної роботи я застосував на практиці метаврестичний алгоритм на прикладі задачі комівояжера. Алгоритм було програмно реалізовано та проведено повний аналіз його роботи з тестуванням виконання на найкращих вхідних параметрах.

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%.