НУЛП, ІКНІ, САПР		Тема	оцінка	підпис
КН-414	1	**АЛГОРИТМ ПОБУДОВИ ДЕРЕВ**		
Коцюба В.С.				
№ залікової: 16081031				
Дискретні моделі в САПР			Викладач:	
			к.т.н., асистент Кривий Р.З.	

Мета:

Вивчення алгоритмів рішення задач побудови остових дерев.

Завдання: Написати програму для побудови мінімального та максимального покриваючого дерева.

Варіант 1. Алгоритм Борувки.

Теоретичні відомості:

Максимальне остове дерево.

Даний зважений неорієнтований граф з вершинами і ребрами. Потрібно знайти таке піддерево цього графа, яке б з'єднувало всі його вершини, і при цьому мало найбільшу можливу вагою (тобто сумою ваг ребер). Таке піддерево називається максимальним остовим деревом.

У природному постановці ця задача звучить наступним чином: є міст, і для кожної пари відома вартість з'єднання їх дорогою (або відомо, що з'єднати їх не можна). Потрібно з'єднати всі міста так, щоб можна було доїхати з будь-якого міста в інший, а при цьому вартість прокладання доріг була б максимальною. Сам алгоритм має дуже простий вигляд. Шуканий максимальний кістяк будується поступово, додаванням до нього ребер по одному. Спочатку остов покладається складається з єдиної вершини (її можна вибрати довільно). Потім вибирається ребро максимальної ваги, що виходить з цієї вершини, і додається в максимальне остове дерево. Після цього остов містить уже дві вершини, і тепер шукається і додається ребро максимальної ваги, що має один кінець в одній з двох обраних вершин, а інший - навпаки, у всіх інших, крім цих двох. І так далі, тобто щоразу шукається максимальне по вазі ребро, один кінець якого - вже взята в остов вершина, а інший кінець - ще не взята, і це ребро додається в остов (якщо таких ребер кілька, можна взяти будь-яке). Цей процес повторюється до тих пір, поки остов не стане містити всі вершини (або, що те ж саме, ребро). У результаті буде побудований остов, що є максимальним . Якщо граф був спочатку не зв'язний, то остов знайдений не буде (кількість вибраних ребер залишиться менше).

Алгоритм Борувки.

Це алгоритм знаходження мінімального остового дерева в графі. Вперше був опублікований в 1926 році Отакаром Борувкой, як метод знаходження оптимальної електричної мережі в Моравії. Робота алгоритму складається з декількох ітерацій, кожна з яких полягає в послідовному додаванні ребер до остового лісу графа, до тих пір, поки ліс не перетвориться на дерево, тобто, ліс, що складається з однієї компоненти зв'язності. У псевдокоді, алгоритм можна описати так: Спочатку, нехай T - порожня множина ребер (представляє собою остовий ліс, до якого кожна вершина входить в якості окремого дерева). Поки T не є деревом (поки число ребер у T менше, ніж V-1, де V - кількість вершин у графі): Для кожної компоненти зв'язності (тобто, дерева в остовому лісі) в підпункті з ребрами T, знайдемо ребро найменшої ваги, що зв'язує цю компоненту з деякої іншої компонентою зв'язності. (Передбачається, що ваги ребер різні, або як-то додатково впорядковані так, щоб завжди можна було знайти єдине ребро з мінімальною вагою). Додамо всі знайдені ребра в множину T. Отримана множина ребер T є мінімальним остовим деревом вхідного графа.

Програмна реалізація основного методу:

```
while (numTree > 1) {
            System.out.println("Number of Vertices:" + numTree);

            //Reset the cheapest values every iteration
            for (int i = 0; i < vertNum; i++) {
                cheapest[i] = -1;
            }

            //Iterate over all edges to find the cheapest
            //edge of every subtree
            for (int i = 0; i < edgeNum; i++) {

                //Find the subsets of the corners of the edge
                int set1 = find(subsets, edges[i].getSrc());
                int set2 = find(subsets, edges[i].getDest());

                //If the two corners belong to the same subset,
                //ignore the current edge
                if (set1 != set2) {

                    //If they belong to different subsets, check which
                    //one is the cheapest
                    if (cheapest[set1] == -1 || edges[cheapest[set1]].getWeight() >
edges[i].getWeight()) {
```

```
                        cheapest[set1] = i;
                    }

                    if (cheapest[set2] == -1 || edges[cheapest[set2]].getWeight() >
edges[i].getWeight()) {
                        cheapest[set2] = i;
                    }
                }
            }

            //Add the cheapest edges obtained above to the MST
            for (int j = 0; j < vertNum; j++) {

                //Check if the cheapest for current set exists
                if (cheapest[j] != -1) {
                    int set1 = find(subsets, edges[cheapest[j]].getSrc());
                    int set2 = find(subsets, edges[cheapest[j]].getDest());

                    if(set1 != set2){
                        MSTweight += edges[cheapest[j]].getWeight();
                        System.out.println("Edge ("+
vertNames[edges[cheapest[j]].getSrc()] + ", " +
vertNames[edges[cheapest[j]].getDest()]+") added to the MST");
                        uniteSubsets(subsets, set1, set2);
                        numTree--;
                    }
                }
            }
        }
     System.out.println("Final weight of MST :" + MSTweight);
```

Результати роботи програми:

Вхідні данні:

```
4 5
0 A
1 B
2 C
3 D
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

Вхідні данні у вигляді текстового файлу, де: перша стрічка від повідає кількості вершин (4) та кількості ребер (5).

```
Initializing Boruvka's MST
Number of Vertices:4
Edge (A, D) added to the MST
Edge (A, B) added to the MST
Edge (C, D) added to the MST
Final weight of MST :19
```

Рис.1. Результат роботи програми

Висновок: На цій лабораторній роботі роботі було здійснено ознайомлення з алгоритмами побудови остових дерев, програмно реалізувано роботу алгоритму Борувки.

НУЛП, ІКНІ, САПР		Тема	оцінка	підпис
КН-414	2	**Алгоритм рішення задачі листоноші**		
Коцюба В.С.				
№ залікової: 16081031				
Дискретні моделі в САПР			Викладач:	
			к.т.н., асистент Кривий Р.З.	

Мета:

Метою даної лабораторної роботи є вивчення і дослідження алгоритмів рішення задачі листоноші.

Завдання:

Написати програму для демонстрації роботи алгоритму задачі листоноші.

Теоретичні відомості:

Задача листоноші. Основні поняття. Властивості.

Будь-який листоноша перед тим, як відправитись в дорогу повинен підібрати на пошті листи, що відносяться до його дільниці, потім він повинен рознести їх адресатам, що розмістились вздовж маршрута його проходження, і повернутись на пошту. Кожен листоноша, бажаючи втратити якомога менше сил, хотів би подолати свій маршрут найкоротшим шляхом. Загалом, задача листоноші полягає в тому, щоб пройти всі вулиці маршрута і повернутися в його початкову точку, мінімізуючи при цьому довжину пройденого шляху.

Перша публікація, присвячена рішенню подібної задачі, появилась в одному з китайських журналів, де вона й була названа задачею листоноші. Очевидно, що така задача стоїть не тільки перед листоношею. Наприклад, міліціонер хотів би знати найбільш ефективний шлях патрулювання вулиць свого району, ремонтна бригада зацікавлена у виборі найкоротшого шляху переміщення по всіх дорогах.

Задача листоноші може бути сформульована в термінах теорії графів. Для цього побудуємо граф G = (X , E), в якому кожна дуга відповідає вулиці в маршруті руху листоноші, а кожна вершина - стик двох вулиць. Ця задача являє собою задачу пошуку найкоротшого маршруту, який включає кожне ребро хоча б один раз і закінчується у початковій вершині руху.

Нехай S-початкова вершина маршруту і a(i,j)>0 - довжина ребра (i, j) . В графі на рис. 1 існує декілька шляхів, по яким листоноша може обійти всі ребра і повернутись у вершину S.

Наприклад :

Шлях 1: (S,a), (a,b), (b,c), (c,d), (d,b), (b,S)

Шлях 2: (S,a), (a,b), (b,d), (d,c), (c,b), (b,S)

Шлях 3: (S,b), (b,c), (c,d), (d,b), (d,a), (a,S)

Шлях 4: (S,b), (b,d), (d,c), (c,b), (b,a), (a,S)

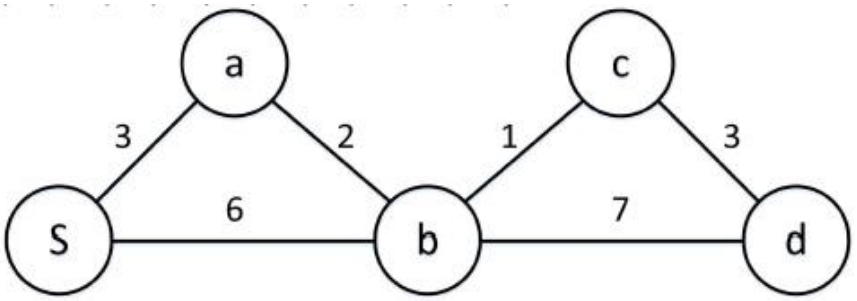

Рис. 1.

В будь-який з чотирьох шляхів кожне ребро входить тільки один раз.

Таким чином, загальна довжина кожного маршруту дорівнює 3+2+1+3+7+6=22.

Кращих маршрутів у листоноші не існує.

Ейлеровий цикл

Ейлеревим циклом в графі називається шлях, який починається і закінчується в тій самій вершині, при чому всі ребра графа проходяться тільки один раз.

Ейлеревим шляхом називається шлях, який починається в вершині А, а закінчується в вершині Б, і всі ребра проходяться лише по одному разу.

Граф, який включає в себе ейлерів цикл називається ейлеревим.

Індивідуальне завдання

Варіант 1. Реалізувати програму для вирішення задачі листоноші.

Робота з програмою:

Після запуску програми в лівому краю вікна, у верхньому текст боксі задана початкова матриця суміжності графу, за потреби її там можна міняти.

Для запуску алгоритму натискаємо кнопку ‘Старт’, у нижньому текст боксі бачимо результати роботи алгоритму у центрі в полі канвас- відображення графа.

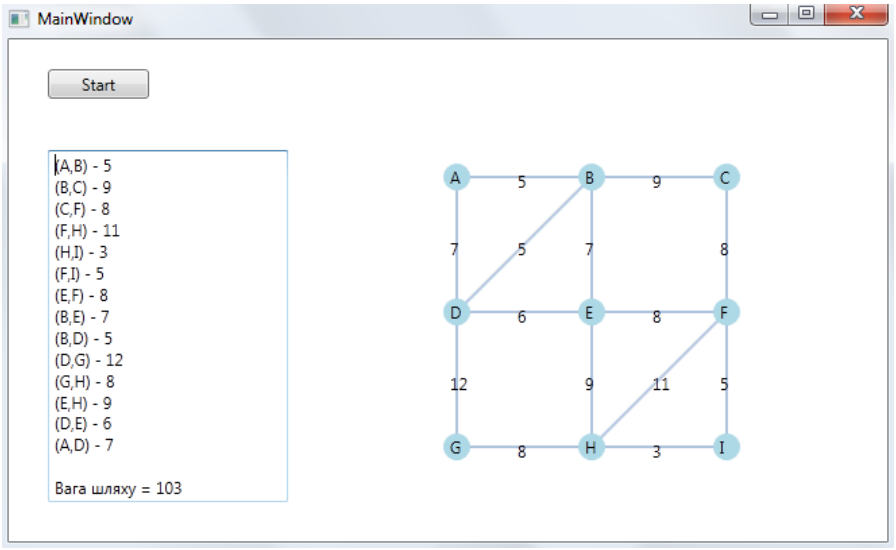

Рис.2 Вікно роботи програми

Фрагмент програми:

```
//зчитати ребра з файлу
        private void ReadEdges(string path, List<Edge> input, List<int> input2)
        {
            String[] str = File.ReadAllLines(path);
            int[,] matrix = new int[str.Length, 3];
            for (int i = 0; i < str.Length; i++)
            {
                int[] arr = str[i].Split(new char[] { ';' }, StringSplitOptions.RemoveEmptyEntries).Select(s =>
int.Parse(s)).ToArray();
                for (int j = 0; j < arr.Length; j++)
                {
                    matrix[i, j] = arr[j];
                }
            }
            for (int i = 0; i < matrix.GetLength(0); i++)
            {
                input.Add(new Edge(matrix[i, 0], matrix[i, 1], matrix[i, 2]));
                input2.Add(matrix[i, 0]);
                input2.Add(matrix[i, 1]);
            }
        }
        //Список вершин
        private List<Vertex> GetUniquesVer(List<int> input)
        {
            Dictionary<int, bool> found = new Dictionary<int, bool>();
            List<Vertex> uniques = new List<Vertex>();
            foreach (int value in input)
            {
                if (!found.ContainsKey(value))
                {
                    found[value] = true;
                    char ch = Convert.ToChar(value + 64);
                    uniques.Add(new Vertex(ch.ToString(), value));
                }
            }
            return uniques;
        }
        //Належність до підграфа
        private void MarkEdge(Edge ed, List<Edge> Elist, int num)
        {
            for (int i = 0; i < Elist.Count; i++)
            {
                if (Elist[i].VERTEX1 == ed.VERTEX1 && Elist[i].VERTEX2 == ed.VERTEX2)
                {
                    ed.COMP = num;
                    break;
                }
            }
        }
        //Дістаю перелік унік. ел.
        private List<int> GetUnique(List<int> input)
        {
            List<int> output = new List<int>();
            Dictionary<int, bool> dict = new Dictionary<int, bool>();
            for (int i = 0; i < input.Count; i++)
            {
```

```
                if (!dict.ContainsKey(input[i]))
                {
                    dict[input[i]] = true;
                    output.Add(input[i]);
                }
            }
            return output;
        }
        private Point[] ReadCoords(string path)
        {
            String[] str = File.ReadAllLines(path);
            Point[] points = new Point[str.Length];
            for (int i = 0; i < str.Length; i++)
            {
                int[] arr = str[i].Split(new char[] { ';' }, StringSplitOptions.RemoveEmptyEntries).Select(s =>
int.Parse(s)).ToArray();
                points[i].X = arr[0];
                points[i].Y = arr[1];
            }
            return points;
        }



        private List<int> CountUniques<T>(List<T> list)
        {
            List<int> result = new List<int>();
            Dictionary<T, int> counts = new Dictionary<T, int>();
            List<T> uniques = new List<T>();
            foreach (T val in list)
            {
                if (counts.ContainsKey(val))
                    counts[val]++;
                else
                {
                    counts[val] = 1;
                    uniques.Add(val);
                }
            }
            foreach (T val in uniques)
            {
                result.Add(counts[val]);
            }
            return result;
        }
        //Словник унікальних
        private Dictionary<T, int> UniquesDict<T>(List<T> list)
        {
            List<int> result = new List<int>();
            Dictionary<T, int> counts = new Dictionary<T, int>();
            List<T> uniques = new List<T>();
            foreach (T val in list)
            {
                if (counts.ContainsKey(val))
                    counts[val]++;
                else
                {
                    counts[val] = 1;
                    uniques.Add(val);
                }
            }

            return counts;
        }
        //Визначення наявності Ейлерового циклу в графі
        private bool IsEilerCycle(List<Vertex> iVer, List<Edge> iEdg)
        {
            bool cycle = true;
            List<int> list = new List<int>();
            for (int i = 0; i < iEdg.Count; i++)
            {
                list.Add(iEdg[i].VERTEX1);
                list.Add(iEdg[i].VERTEX2);
            }
            list = CountUniques<int>(list);
            foreach (int item in list)
            {
                if (item % 2 != 0)
                {
                    cycle = false;
                    break;
                }
            }

            return cycle;
                 }
```

Висновок: На цій лабораторній роботі було здійснено ознайомлення з алгоритмом рішення задачі листоноші.

НУЛП, ІКНІ, САПР		Тема	оцінка	підпис
КН-414	3	**Потокові алгоритми**		
Коцба В.С.				
№ залікової: 16081031				
Дискретні моделі в САПР			Викладач:	
			к.т.н., асистент Кривий Р.З.	

Мета:

Вивчення потокових алгоритмів.

Завдання: Написати програму для демонстрації роботи обраного потокового алгоритму.

Варіант 13. Алгоритм Форда-Фалкерсона.

Теоретичні відомості:

Алгоритм Форда-Фалкерсона є одним з способів рішення задачі побудови максимального потоку в мережі.

Опис:

Знайти будь-який шлях, що збільшується. Збільшити потік по всіх його ребрах на мінімальну з їх залишкових пропускних здатностей. Повторювати, поки є шлях, що збільшується. Алгоритм працює тільки для цілих пропускних здатностей. В іншому випадку, він може працювати нескінченно довго, не сходячись до правильної відповіді.

Складність:

Залежить від алгоритму пошуку шляху, що збільшується. Потребує O(max| f |) таких пошуків.

Для пошуку шляху, що збільшується я використав алгоритм пошуку в ширину.

Алгоритм має таке практичне значення: рішення транспортної задачі, наприклад, потрібно перевезти з початкової вершини мережі в кінцеву вантаж по дугах мережі за мінімальний час, при цьому по кожній дузі не можна перевозити вантажу більше фіксованого об’єму.

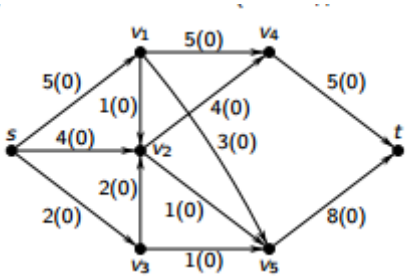

Рис.1 Початковий граф

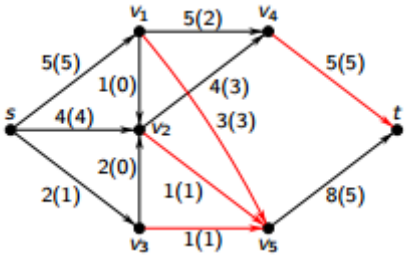

Рис.2 Граф після роботи алгоритму (макс. потік 10)

Робота з програмою:

Для запуску роботи алгоритму потрібно нажати кнопку ‘Старт’.

Для вводу всіх точок записаних у програму зразу використовується кнопка ‘Додати всі’.

Для очищення списку ребер та графу потрібно нажати ‘Очистити’.

Для додавання ребер по одному вводимо на панелі додавання відповідні дані (дві вершини та вагу ребра) та нажимаємо ‘Додати’.

На панелі задання початку та кінця задаємо номери початкової та кінцевої вершин.

У верхньому текст боксі відображається результат роботи алгоритму, а у нижньому виводиться інформація про роботу програми.

Справа на елементі Канвас відображається побудований граф.

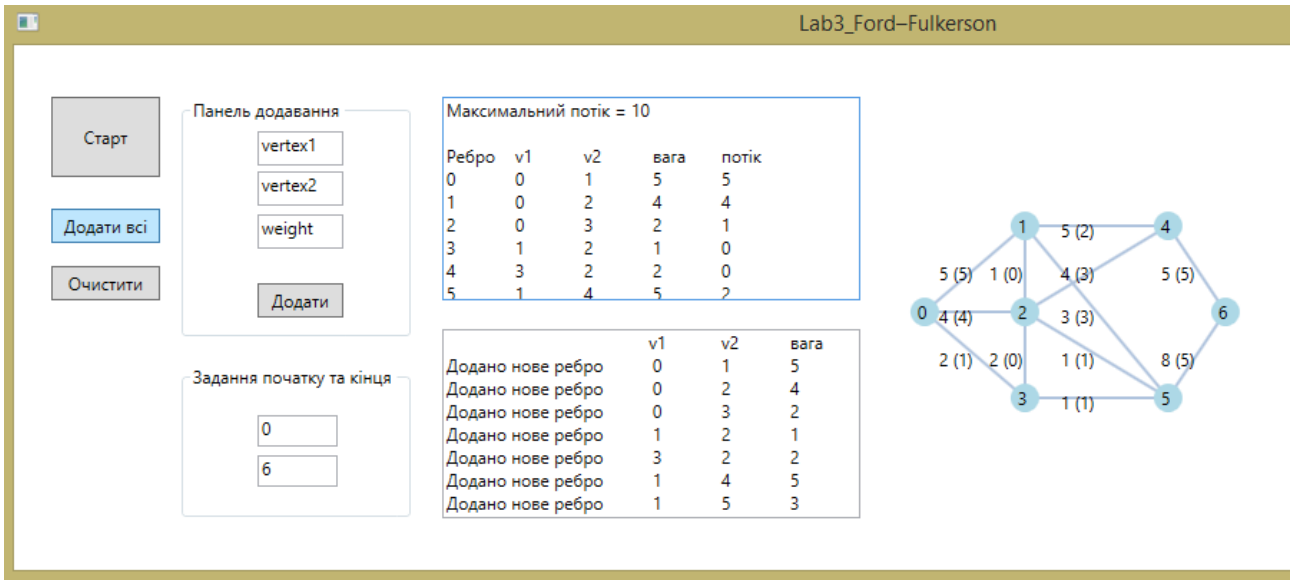

Рис.3 Результат роботи програми

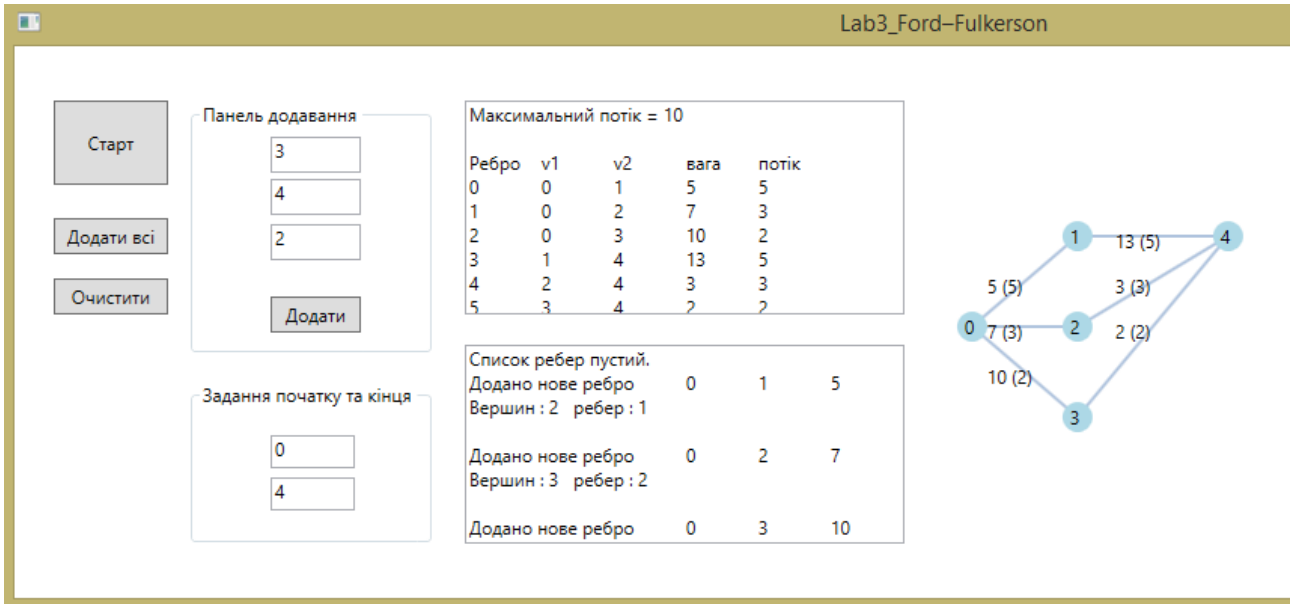

Рис.4 Результат роботи програми для інших даних

Фрагмент програми:

```
int FordFulkerson(int source, int sink) {
     int i, j, u;
     int max_flow = 0;
     for (i = 0; i < nodes; i++) {
         for (j = 0; j < nodes; j++) {
             flow[i, j] = 0;
         }
     }
     while (bfs(source,sink)) {
         int increment = 10000;
         for (u = nodes - 1; pred[u] >= 0; u = pred[u]) {
             increment = min(increment, capacity[pred[u], u] - flow[pred[u], u]);
         }
         for (u = nodes - 1; pred[u] >= 0; u = pred[u]) {
             flow[pred[u], u] += increment;
             flow[u, pred[u]] -= increment;
         }
         max_flow += increment;

     }
     return max_flow;
        }
```

Висновок: На цій лабораторній роботі було здійснено ознайомлення з потоковим алгоритмом Форда-Фалкерсона та програмно реалізовано його.

НУЛП, ІКНІ, САПР		Тема	оцінка	підпис
КН-414	4	**АЛГОРИТМ РІШЕННЯ ЗАДАЧІ КОМІВОЯЖЕРА**		
Коцюба В.С.				
№ залікової: 16081031				
Дискретні моделі в САПР			Викладач:	
			к.т.н., асистент Кривий Р.З.	

Мета:

Метою даної лабораторної роботи є вивчення і дослідження алгоритмів рішення задачі комівояжера.

Завдання:

Написати програму для демонстрації роботи алгоритму задачі комівояжера.

Теоретичні відомості:

Умови існування гамільтонового контуру. Нижні границі.

Рішенням задачі комівояжера є оптимальний гамільтоновий контур. Нажаль, не всі графи містять гамільтоновий контур. Отже перед тим, ніж перейти до пошуку оптимального гамільтонового контура потрібно довести факт його існування в даному графі.

Можна знайти точний розв'язок задачі комівояжера, тобто, обчислити довжини всіх можливих маршрутів та обрати маршрут з найменшою довжиною. Однак, навіть для невеликої кількості міст в такий спосіб задача практично нерозв'язна. Для простого варіанта, симетричної задачі з n містами, існує $(n - 1)! / 2$ можливих маршрутів, тобто, для 15 міст існує 43 мільярдів маршрутів та для 18 міст вже 177 більйонів. Те, як стрімко зростає тривалість обчислень можна показати в наступному прикладі. Якщо існував би пристрій, що знаходив би розв'язок для 30 міст за годину, то для для двох додаткових міст в тисячу раз більше часу; тобто, більш ніж 40 діб.

Відомо багато різних методів рішення задачі комівояжера. Серед них можна виділити методи розроблені Белмором і Немхаузером, Гарфинкелем і Немхаузером, Хелдом і Карном, Стекханом. Всі ці методи відносяться до одного з двох класів: а) методи рішення, які завжди приводять до знаходження оптимального рішення, але потребують для цього, в найгіршому випадку, недопустимо великої кількості операцій(метод гілок та границь); б) методи, які не завжди приводять до находження оптимального результату, але потребують для цього допустимої великої кількості операцій (метод послідовного покращення рішення).

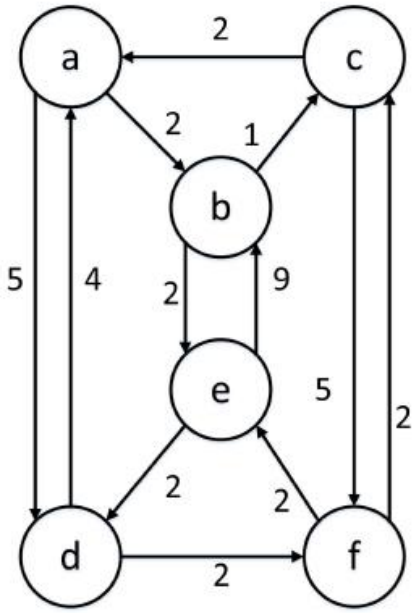

Рис.1 Заданий граф

Робота з програмою:

Після запуску програми в лівому краю вікна, у верхньому текст боксі задана початкова матриця суміжності графу, за потреби її там можна міняти.

Для запуску алгоритму натискаємо кнопку ‘Старт’, у нижньому текст боксі бачимо результати роботи алгоритму (існує чи не існує маршрут), у центрі в полі канвас-відображення графа, а у правому текст боксі описані ребра, тобто вершини ребер та їх вага.

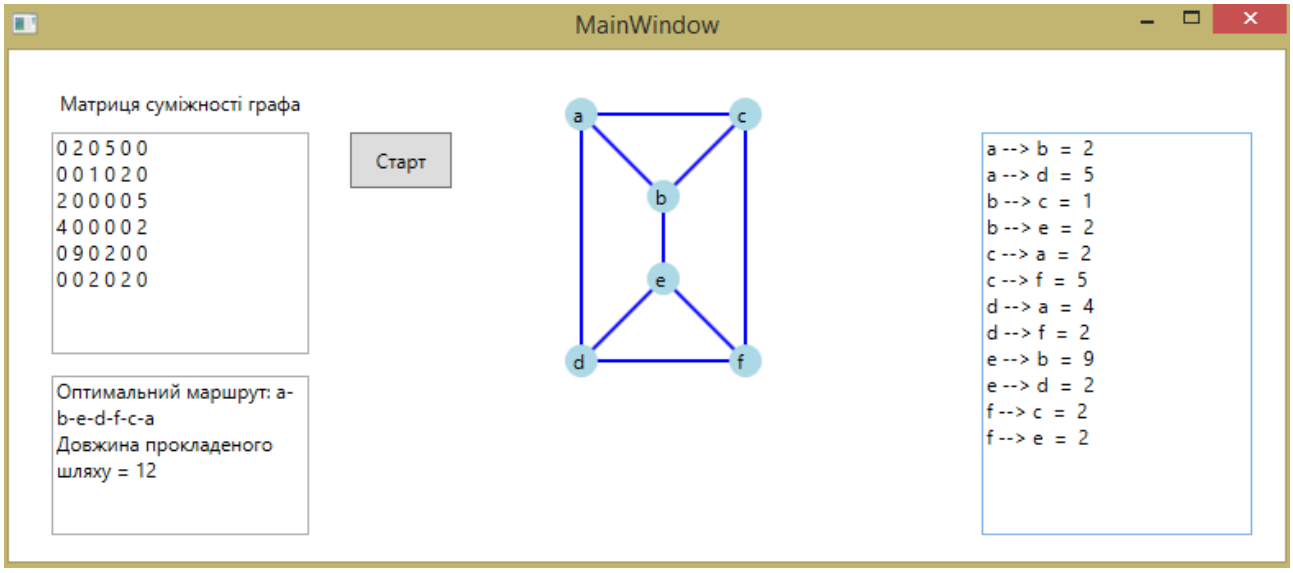

Рис.2 Вікно роботи програми

Фрагмент програми:

```
public String route(string alphabet)
        {
            String res = "";
            for (int j = 1; j <= n; j++)
                current_way[j] = 0;
            sum = 0;//занулюємо суму
            end_way = 0;//шлях вважаємо не знайденим
            passedVertex = 1;
```

```
            current_way[1] = passedVertex;                      //починаємо обхід з першої вершини
            minSum = Int32.MaxValue / 2;
            recursSearch(1);                                //шукаємо починаючи з першої вершини

            if (end_way > 0)                                    //якщо знайдено шлях
            {
                res += "Оптимальний маршрут: ";             //починаємо формувати результуючу стрічку

                int c = 1;                                      //номер в порядку обходу вершин

                for (int i = 1; i <= n; i++)                    //проходимо по всіх вершинах
                {
                    int j = 1;
                    while ((j <= n) && (minWay[j] != c))        //шукаємо наступну вершину в порядку обходу
                        j++;

                    res += alphabet[j - 1] + "-";    //додаємо вершину до результуючої стрічки
                    c++;
                }

                res += alphabet[0];//до результуючої стрічки додаємо першу вершину, якою завершується обхід
                res += "\nДовжина прокладеного шляху = " + minSum;//до результуючої стрічки додаємо суму ваг
пройдених ребер

            }
            else
                res = "Не вдалося знайти шлях.";


            return res;
        }

        private void recursSearch(int x)
        {
            //якщо всі вершини переглянуті,
            //і з останньої вершини є шлях в першу,
            //і мнова сума відстаней менша мінімальної
            if ((passedVertex == n) && (matrix_for_computations[x, 1] != 0) && (sum +
matrix_for_computations[x, 1] < minSum))
            {
                end_way = 1;                                //шлях вважається знайденим
                minSum = sum + matrix_for_computations[x, 1];    //вводимо нову мінімальну суму відстаней
                for (int i = 1; i <= n; i++)
                    minWay[i] = current_way[i];
        //вводимо новий мінімальний шлях
            }
            else
            {

                for (int i = 1; i <= n; i++)                    //переглядаємо всі вершини з поточної

                    //нова вершина не співпадає з біжучою, є прямий шлях з біжучої вершини в нову,
                    //нова вершина ще не переглянута, нова сума є меншою за мінімальну
                    if ((i != x) && (matrix_for_computations[x, i] != 0) && (current_way[i] == 0) && (sum +
matrix_for_computations[x, i] < minSum))
                    {
                        sum += matrix_for_computations[x, i];     //збільшуємо суму
                        passedVertex++;                         //збільшуємо кількість переглянутих вершин
                        current_way[i] = passedVertex;  //відмічаємо у нової вершини новий номер у порядку
обходу
                      recursSearch(i);          //пошук нової вершини починаючи з і (вершина, в яку перейшли)
                        current_way[i] = 0;                     //повертаємо все назад
                        passedVertex--;
                        sum -= matrix_for_computations[x, i];
                    }
            }
                }
```

Висновок: На цій лабораторній роботі було здійснено ознайомлення з алгоритмом рішення задачі комівояжера.

НУЛП, ІКНІ, САПР		Тема	оцінка	підпис
КН-414	5	**Ізоморфізм графів**		
Коцюба В.С.				
№ залікової: 16081031				
Дискретні моделі в САПР			Викладач:	
			к.т.н., асистент Кривий Р.З.	

Мета:

Вивчення і дослідження основних підходів до встановлення ізоморфізму графів.

Завдання:

Реалізувати метод повного перебору для встановлення ізоморфізму графів

Теоретичні відомості:

Два графа G=(X,U,P) і G'=(X',U',P') називаються ізоморфними, якщо між їх вершинами, а також між їхніми ребрами можна встановити взаємно однозначне співвідношення X <-> X', U <-> U', що зберігає інцидентність, тобто таке, що для всякої пари (x,y)єX ребра u є U, що з'єднує їх, обов'язково існує пара (x',y') є X' і ребро u' є U', що з'єднує їх, і навпаки. Тут P - предикат, інцидентор графа G. Зауважимо, що відношення ізоморфізму графів рефлексивне, симетричне і транзитивне, тобто представляє собою еквівалентність.

Одним з найпростіших з точки зору програмної реалізації, є алгоритм перевірки ізоморфізму графів повним перебором(можливої перенумерації вершин), але складність цього алгоритму є факторіальною.

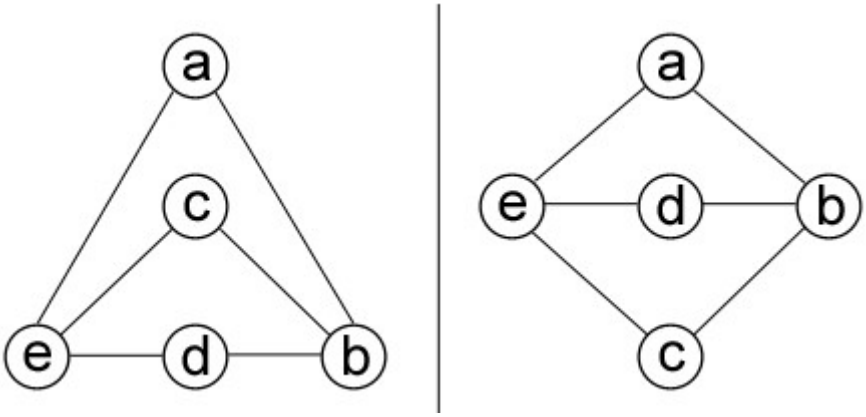

Рис.1 Графи для перевірки ізоморфізму (5 вершин)

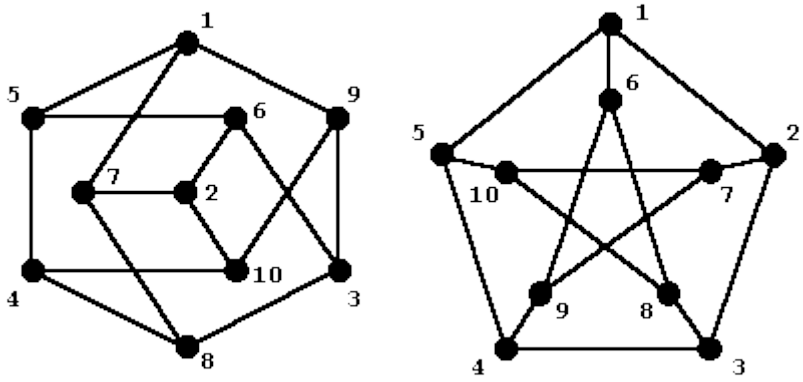

Рис.2 Графи для перевірки ізоморфізму (10 вершин)

Робота з програмою:

Після запуску програми у лівому краї вікна відображаються 2 текстбокса з матрицями суміжності графів.

Вводимо кількість вершин у відповідному місці.

Нажимаємо кнопку 'Запуск алгоритму', у центральному нижньому текстбоксі з'явиться інформація про те, чи є графи ізоморфними.

У правому краї екрану на двох канвасах відображаються самі графи.

Якщо виникає потреба змінити матриці суміжності, то міняємо потрібні цифри в текстбоксах та знову натискаємо кнопку 'Запуск алгоритму'.

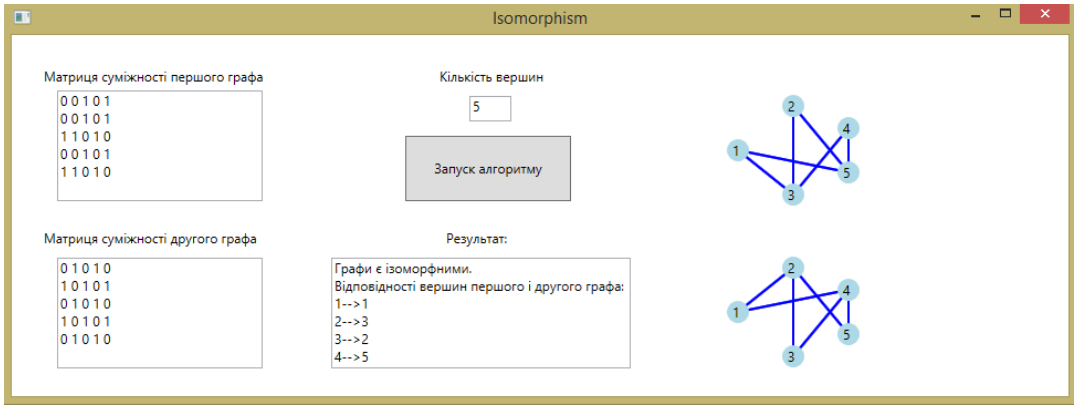

Рис.3 Графи ізоморфні (5 точок)

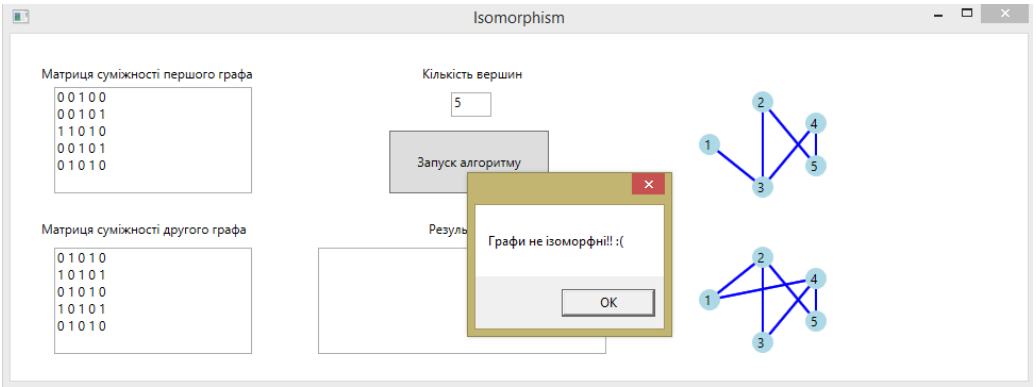

Рис.4 Графи не ізоморфні (після зміни елемента матриці)

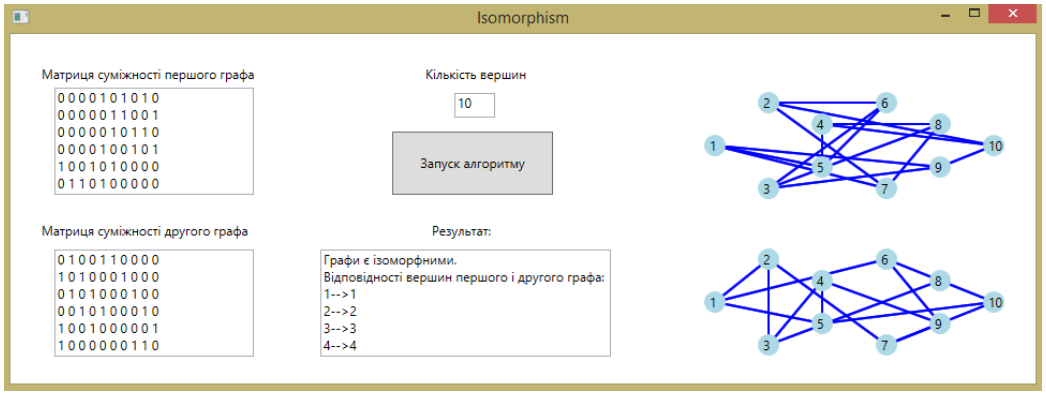

Рис.5 Результат роботи програми, графи ізоморфні (для 10 точок)

Фрагмент програми:

```
  public void BrutalForce()
            {

                firstSumArray = new int[numVert];
                secondSumArray = new int[numVert];
                conformity = new int[numVert];

                for (int i = 0; i < numVert; i++)//перебираємо всі вершини
                    conformity[i] = -1;//ставимо їм у відповідність вершину -1

                Draw(canv1);
                Draw(canv2);
                tb4.Text = Res();

                   }
public String Res
            {
                for (int i = 0; i < numVert; i++)//перебираємо усі рядки
                {
                    for (int j = 0; j < numVert; j++)//перебираємо усі вершини
                    {
                        firstSumArray[i] += data1[i, j];//додаємо вагу ребра
                        secondSumArray[i] += data2[i, j];//додаємо вагу ребра
                    }
                }

                for (int i = 0; i < numVert; i++)//перебираємо усі вершини
                {
                    for (int j = 0; j < numVert; j++)//перебираємо усі вершини
                    {
                        bool isDone = false;//чи перевірки завершені

                        for (int k = 0; k < numVert; k++)//перебираємо усі вершини
                            if (conformity[k] == j)//чи вершина має відповідну
                                isDone = true;//перевірки завершено

                        if (!isDone && firstSumArray[i] == secondSumArray[j])//якщо перевірки не
завершені і вершина має відповідну
                        {
                            conformity[i] = j;//встановлюємо поточне j як відповідну вершину
                            break;//виходимо з циклу
                        }

                    }
                }

                result = "Графи є ізоморфними.";//формуємо стрічку-відповідь
                bool isrouted = true; //чи було знайдено розвязок

                for (int i = 0; i < numVert; i++)//перебираємо усі вершини
```

```
                    if (conformity[i] == -1)//якщо для вершини нема відповідної
                    {
                        MessageBox.Show("Графи не ізоморфні!! :(");
                        isrouted = false;//рішення не знайдено
                        result = "Графи не є ізоморфними.";//формуємо стрічку-відповідь
                        break;//виходимо з циклу
                    }

                if (isrouted)//якщо рішення знайдено
                {
                    MessageBox.Show("Графи ізоморфні!! :)");
                    result += "\nВідповідності вершин першого і другого графа: \n";//доповнюємо
стрічку-відповідь
                    for (int i = 0; i < numVert; i++)//перебираємо всі вершини
                    {
                        result += (i + 1) + "-->" + (conformity[i] + 1) + " ";//додаємо до
стрічки відповіді відповідності вершин
                        result += "\n";
                    }
                }

                return result;//повертаємо отриману стрічку-відповідь
                   }
```

Висновок: На цій лабораторній роботі було здійснено ознайомлення з алгоритмом повного перебору визначення ізоморфності графів.

Програмні реалізації лабораторних робіт: https://github.com/vetalztr/DM_Labs