Факультет Математики и Информатики  
Департамент Информатики

**Лабораторная работа №1**

По курсу “Алгоритмика графов”

Тема:”Алгоритмы DFS и BFS”

Выполнил студент группы I1902:  
Чобану Артём

Проверил:

Кэлину Цуркану

Кишинэу, 2020

Проект можно скачать по ссылке(Код тоже в отчёте): https://github.com/ArtiomCiobanu/GraphsTest

**Вариант 4**

Задание:

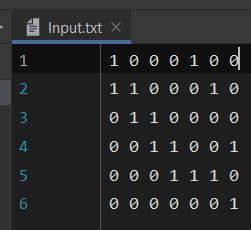
1. Вводится граф через матрицу инцидентности затем, из нее, составить матрицу Киргова, затем, из нее, составить матрицу смежности а затем составить алгебрическое представление графа.
2. Дается простой, несвязанный граф *G*. Используя прохождение в ширину найти количество связанных компонентов графа.

**Матрица инцидентности** – это бинарная матрица, в которой строки соответствуют вершинам, а столбцы – рёбрам, на пересечении которых находится число “1” если ребро инцидентно матрице и “0” – в противном случае.

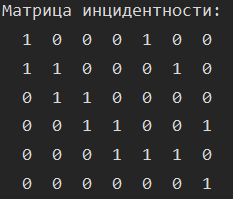
**Матрица Кирхгофа** – это квадратная матрица, в которой строки и столбцы соответствуют вершинам, на главной диагонали которой находится степень вершины, а вне главной диагонали - “-1” если вершины смежны, и “0” – в противном случае.

**Матрица смежности** – это бинарная квадратная матрица, в которой строки и столбцы соответствуют вершинам, на пересечении которых находится “1” ли вершины смежны и “0” - если нет.

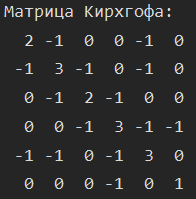
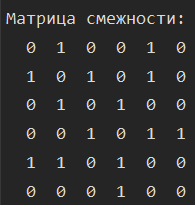
В моём проекте матрица инцидентности считывается из текстового файла:



Вывод:



Далее происходит преобразование в матрицы Кирхгофа и смежности:

Код метода, который преобразовывает матрицу инцидентности в матрицу Кирхгофа:

public static int[][] GetKirchhoffFromIncidence(int[][] incidenceMatrix)  
{  
 var rowAmount = incidenceMatrix.Length;  
 var colAmount = incidenceMatrix[0].Length;  
 int[][] kirchoff = Tools.GetEmptyJuggedArray(rowAmount);  
 for (int i = 0; i < rowAmount; i++)  
 {  
 int deg = incidenceMatrix[i].Sum();  
 kirchoff[i][i] = deg;  
 }  
 for (int c = 0; c < colAmount; c++)  
 {  
 int i = -1, j = -1;  
 for (int r = 0; r < rowAmount; r++)  
 {  
 if (incidenceMatrix[r][c] == 1)  
 {  
 if (i == -1)  
 {  
 i = r;  
 } else  
 {  
 j = r;  
 break;  
 }  
 }  
 }

kirchoff[i][j] = -1;  
 }  
 for (int i = 0; i < rowAmount; i++)  
 {  
 for (int j = 0; j < i; j++)  
 {  
 kirchoff[i][j] = kirchoff[j][i];  
 }  
 }  
 return kirchoff;  
}

Код метода, который преобразовывает матрицу Кирхгофа в матрицу смежности:

public static int[][] GetAdjacencyFromKircoff(int[][] kirchhoff)  
{  
 var rowAmount = kirchhoff.Length;  
 var colAmount = kirchhoff[0].Length;  
  
 int[][] adjacencyMatrix = Tools.GetEmptyJuggedArray(rowAmount);  
  
 for (int i = 0; i < rowAmount; i++)  
 {  
 int deg = kirchhoff[i][i];  
 int currentDeg = 0;  
 for (int j = 0; j < rowAmount; j++)  
 {  
 if (kirchhoff[i][j] == -1)  
 {  
 adjacencyMatrix[i][j] = 1;  
 currentDeg++;  
 }  
 }  
  
 if (currentDeg < deg)  
 {  
 adjacencyMatrix[i][i] = 1;  
 }  
 }  
  
 return adjacencyMatrix;  
}

Код метода Main:

private static void Main(string[] args)  
{  
 string input = File.ReadAllText(@"Input.txt");  
 IncidenceMatrix = Tools.GetIntArrayFromString(input);  
  
 Console.WriteLine("Матрица инцидентности:");  
 Tools.PrintIntMatrix(IncidenceMatrix);  
  
 KirchhoffMatrix = GraphTools.GetKirchhoffFromIncidence(IncidenceMatrix);  
 Console.WriteLine("Матрица Кирхгофа:");  
 Tools.PrintIntMatrix(KirchhoffMatrix);  
  
 AdjacencyMatrix = GraphTools.GetAdjacencyFromKircoff(KirchhoffMatrix);  
 Console.WriteLine("Матрица смежности:");  
 Tools.PrintIntMatrix(AdjacencyMatrix);

Graph graph1 = new Graph(AdjacencyMatrix);  
 Console.WriteLine("Смежные вершины:");  
 graph1.PrintAdjacencyMatrix();

Console.WriteLine("DFS:");  
 graph1.Dfs(0);  
 Console.WriteLine("BFS: обход");  
 graph1.Bfs(0);  
 Console.WriteLine("BFS: поиск компонентов");  
 graph1.BfsComponentSearch(0);  
  
 Console.WriteLine("Алгебраическое представление графа:");  
 GraphTools.PrintAlgebraicRepresentation(IncidenceMatrix);  
}

Я создал класс **Graph**, представляющий собой граф, в котором есть:

* Свойство **VertexAmount** – количество вершин
* Свойство **Vertices** – список вершин
* Метод **PrintAdjacencyMatrix**, который выводит список вершин и смежные и вершины
* Метод **MakeVerticesNotVisited**, устанавливающий флаг всех вершин на false – т.е. не пройденные
* **Dfs** – проходит граф по алгоритму Dfs
* **Bfs** - проходит граф по алгоритму Bfs, а также
* **BfsComponentSearch** – поиск связных компонентов по алгоритму Bfs

Граф инициализируется с помощью матрицы смежности, которая передаётся в конструктор.

Класс **Vertex** представляет собой вершину графа. В нём есть:

* Свойство **IsVisited** – флаг, нужный для того, чтобы пометить вершину как посещённую
* **AdjecentVertices** – список смежных вершин(индексов)
* Index – номер вершины
* Методы **SetIsVisited** и **SetIsNotVisited** для установки/снятия флага
* Метод **IsAdjacentTo** для проверки смежности с конкретной вершиной

Кроме того, есть вспомогательные статические классы:

* **GraphTools** – методы для обработки графов
* **Tools** – общие вспомогательные методы(вывод матриц и т.д.)

Класс **Vertex**:

public class Vertex  
{  
 public bool IsVisited { get; private set; }  
  
 public List<int> AdjecentVertices { get; set; }  
  
 public int Index { get; set; }  
  
 public Vertex(List<int> adjecentVertices, int index)  
 {  
 AdjecentVertices = adjecentVertices;  
 Index = index;  
 }  
  
 public void SetIsVisited()  
 {  
 IsVisited = true;  
 }  
  
 public void SetIsNotVisited()  
 {  
 IsVisited = false;  
 }  
  
 public bool IsAdjacentTo(Vertex vertex)  
 {  
 return AdjecentVertices.Contains(vertex.Index);  
 }  
}

Класс **Graph**:

public class Graph  
{  
 private int VertexAmount { get; set; }  
  
 private List<Vertex> Vertices { get; set; }  
  
 public Graph(int[][] adjacencyMatrix)  
 {  
 VertexAmount = adjacencyMatrix.Length;  
  
 Vertices = new List<Vertex>();  
 for (int i = 0; i < VertexAmount; i++)  
 {  
 var adjacencyList = new List<int>();  
 for (int j = 0; j < VertexAmount; j++)  
 {  
 if (adjacencyMatrix[i][j] == 1)  
 {  
 adjacencyList.Add(j);  
 }  
 }  
  
 Vertices.Add(new Vertex(adjacencyList, i));  
 }  
 }  
  
 public void PrintAdjacencyMatrix()  
 {  
 for (var i = 0; i < VertexAmount; i++)  
 {  
 Console.Write($"{i}:");  
 foreach (var j in Vertices[i].AdjecentVertices)  
 {  
 Console.Write($"{j}".PadLeft(3));  
 }  
  
 Console.WriteLine();  
 }  
 }  
  
 private void MakeVerticesNotVisited()  
 {  
 foreach (var vertex in Vertices)  
 {  
 vertex.SetIsNotVisited();  
 }  
 }

Обход DFS:

public void Dfs(int firstVertexIndex)  
{  
 var first = Vertices[firstVertexIndex];  
 first.SetIsVisited();  
  
 var stack = new Stack<Vertex>();  
 stack.Push(first);  
  
 while (stack.Count != 0)  
 {  
 var current = stack.Pop();  
 Console.Write($"{current.Index} ");  
  
 foreach (var adjIndex in current.AdjecentVertices)  
 {  
 current = Vertices[adjIndex];  
 if (!current.IsVisited)  
 {  
 current.SetIsVisited();  
 stack.Push(current);  
 }  
 }  
 }  
  
 Console.WriteLine();  
 MakeVerticesNotVisited();  
}

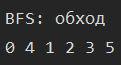
Вывод вершин:



Обход BFS:

public void Bfs(int firstVertexIndex)  
{  
 var first = Vertices[firstVertexIndex];  
  
 var stack = new Stack<Vertex>();  
 stack.Push(first);  
  
 while (stack.Count != 0)  
 {  
 var queuedVertices = new List<Vertex>();  
 var currentVertexCount = stack.Count;  
 for (var i = 0; i < currentVertexCount; i++)  
 {  
 var current = stack.Pop();  
 current.SetIsVisited();  
 Console.Write($"{current.Index} ");  
  
 foreach (var adjIndex in current.AdjecentVertices)  
 {  
 var vertex = Vertices[adjIndex];  
 if (!vertex.IsVisited && !stack.Contains(vertex))  
 {  
 queuedVertices.Add(vertex);  
 }  
 }  
 }  
  
 stack = new Stack<Vertex>(queuedVertices);  
 }  
  
 Console.WriteLine();  
 MakeVerticesNotVisited();  
}

Вывод:

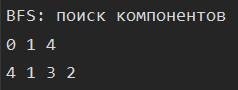


Поиск связных компонентов алгоритмом BFS:

public void BfsComponentSearch(int firstVertexIndex)  
{  
 var first = Vertices[firstVertexIndex];  
  
 var stack = new Stack<Vertex>();  
 stack.Push(first);  
  
 while (stack.Count != 0)  
 {  
 List<Vertex> leftChain = stack.ToList();  
 var queuedVertices = new List<Vertex>();  
 var currentVertexCount = stack.Count;  
 for (var i = 0; i < currentVertexCount; i++)  
 {  
 var current = stack.Pop();  
 current.SetIsVisited();  
 foreach (var adjIndex in current.AdjecentVertices)  
 {  
 var vertex = Vertices[adjIndex];  
 if (!vertex.IsVisited && !stack.Contains(vertex))  
 {  
 queuedVertices.Add(vertex);  
 }  
 }  
 }  
 if (GraphTools.IsAChain(queuedVertices) &&

GraphTools.ChainsAreConnected(leftChain, queuedVertices))  
 {  
 foreach (var v in leftChain)  
 {  
 Console.Write($"{v.Index} ");  
 }  
  
 foreach (var v in queuedVertices)  
 {  
 Console.Write($"{v.Index} ");  
 }  
  
 Console.WriteLine();  
 }  
 stack = new Stack<Vertex>(queuedVertices);  
 }  
 MakeVerticesNotVisited();  
}

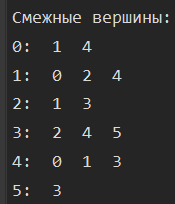
Вывод:



Вывод смежных вершин графа:

public void PrintAdjacencyMatrix()  
{  
 for (var i = 0; i < VertexAmount; i++)  
 {  
 Console.Write($"{i}:");  
 foreach (var j in Vertices[i].AdjecentVertices)  
 {  
 Console.Write($"{j}".PadLeft(3));  
 }  
  
 Console.WriteLine();  
 }  
}

Вывод:



Алгебраическое представление графа:

public static void PrintAlgebraicRepresentation(int[][] incidenceMatrix)  
{  
 Console.Write("{ ");  
  
 var rowAmount = incidenceMatrix.Length;  
 var colAmount = incidenceMatrix[0].Length;  
  
 for (int j = 0; j < colAmount; j++)  
 {  
 int v1 = -1, v2 = -1;  
 for (int i = 0; i < rowAmount; i++)  
 {  
 if (incidenceMatrix[i][j] == 1)  
 {  
 if (v1 == -1)  
 {  
 v1 = i;  
 }  
 else  
 {  
 v2 = i;  
 break;  
 }  
 }  
 }  
  
 Console.Write($"({v1},{v2}) ");  
 }  
  
 Console.Write("}");  
}

Вывод:

