

Exercises: Graphs and Graph Algorithms

Problems for exercises and homework for the "Data Structures and Algorithms Basics" course from the official "Applied Programmer" curriculum.

You can check your solutions here: <https://judge.softuni.bg/Contests/2916/Graphs-Exercises>

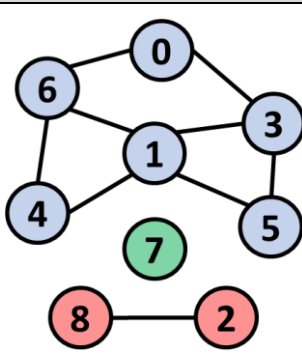

1. Connected Components

The first part of this exercise aims to implement the **DFS algorithm** (Depth-First-Search) to **traverse a graph** and find its **connected components** (nodes connected to each other either directly, or through other nodes). The graph nodes are numbered from **0** to **n-1**. The graph comes from the console in the following format:

- First line: number of lines **n**
- Next **n** lines: list of child nodes for the nodes **0 ... n-1** (separated by a space)

Print the connected components in the same format as in the examples below:

Example

Input	Graph	Output
9 3 6 3 4 5 6 8 0 1 5 1 6 1 3 0 1 4 2		Connected component: 6 4 5 1 3 0 Connected component: 8 2 Connected component: 7
1 0		Connected component: 0

DFS Algorithm

First, create a **boolean array** that will be with the size of your graph.

```
private static bool[] visited;
```

Next, implement the **DFS algorithm** (Depth-First-Search) to traverse all nodes connected to the specified start node:

```
private static void DFS(int vertex)
{
    if (!visited[vertex])
    {
        visited[vertex] = true;
        foreach (var child in graph[vertex])
        {
            DFS(child);
        }
        Console.Write(" " + vertex);
    }
}
```

Find All Components

We want to **find all connected components**. We can just run the DFS algorithm for each node taken as a start (which was not visited already):

```
private static void FindGraphConnectedComponents()
{
    visited = new bool[graph.Length];

    for (int startNode = 0; startNode < graph.Length; startNode++)
    {
        if (!visited[startNode])
        {
            Console.Write("Connected component:");
            DFS(startNode);
            Console.WriteLine();
        }
    }
}
```

Read Input

Let's implement the data entry logic (**read graph** from the console). We already have the method below:

```
private static List<int>[] ReadGraph()
{
    int n = int.Parse(Console.ReadLine());
    var graph = new List<int>[n];
    for (int i = 0; i < n; i++)
    {
        graph[i] = Console.ReadLine()
            .Split(" ", StringSplitOptions.RemoveEmptyEntries)
            .Select(int.Parse)
            .ToList();
    }

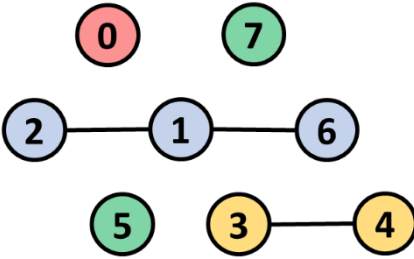
    return graph;
}
```

The main method should **read the graph from the console**:

```
static void Main()
{
    graph = ReadGraph();
    FindGraphConnectedComponents();
}
```

Now test the program. Run it by **[Ctrl] + [F5]**. Enter a sample graph data and check the output:

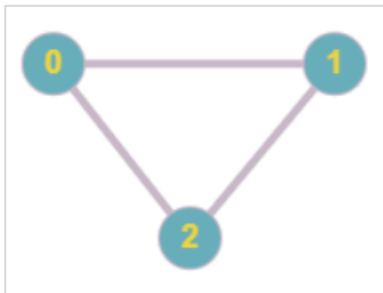
Input	Graph	Expected Output
-------	-------	-----------------

7		Connected component: 0 Connected component: 2 6 1 Connected component: 4 3 Connected component: 5 Connected component: 7
---	---	--

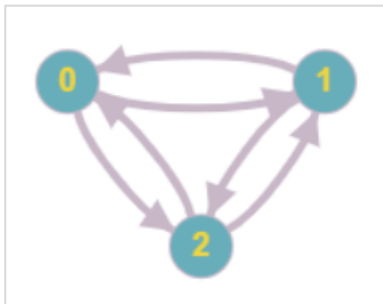
Test with other examples from above, as well.

2. Cycles in a Graph

Our task is to find if there is a **cycle** in a given **undirected graph**, using the **DFS** algorithm. An **undirected graph** is graph, where all the **edges are bidirectional**. For example, in the graph shown below there is an undirected edge between vertex 0 and vertex 1 – this means that there is a directed edge from vertex 0 to vertex 1 and from vertex 1 to vertex 0.




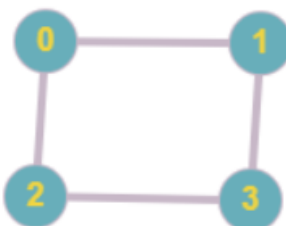
In this case, if we create a **directed graph** from the undirected one shown above, it will look like this:



Now, let's find out how to check whether the graph contains a **cycle** (a non-empty trail in which the only repeated vertices are the first and last vertices). For more clearance, look at the examples below.

Examples

Input	Picture	Output
2 1 0-1		Graph doesn't contain cycle

4 4 0-1 1-2 2-3 0-2		Graph contains cycle
------------------------------------	---	----------------------

Create Graph

First, create a class **Graph** with **fields** for the count of vertices and for the adjacents. You should also create a **constructor** for the Graph initialization. We have done this before and you should know how to do it. If you have difficulties, look for instructions in "**Graphs-Lab.docx**" document.

Read the Graph

In this task, we need to **read** the graph with its vertices and edges from the console. Create a **ReadGraph()** method and use it to initialize the graph from the **Main()**. The method should read the count of vertices, the count of edges and the edges (the first vertex and the second vertex are separated by "-") from the console. We have done this as well in the "**Graphs-Lab.docx**".

Note that our **ReadGraph()** method should invoke the **AddEdge()** method for adding edges to the graph. However, it is a bit different when we have an **undirected** graph. In the **AddEdge()** method, we should create an edge from the first to the second vertex and from the second to the first one (this way we have an undirected edge). The method should look like this:

```
public void AddEdge(int firstVertex, int secondVertex)
{
    adjacents[firstVertex].Add(secondVertex);
    adjacents[secondVertex].Add(firstVertex);
}
```

Create the Algorithm

First, create an **isCyclic()** method, which returns a **boolean** (true/false whether there is a cycle or not). In it, create a **boolean array** for visited vertices and mark all vertices as **not visited**. Then, loop through vertices and invoke a **helper method** to detect cycle in different DFS trees for the ones, which have not been visited:

```
private bool isCyclic()
{
    bool[] visited = new bool[verticesCount];
    for (int i = 0; i < verticesCount; i++)
    {
        visited[i] = false;
    }

    for (int u = 0; u < verticesCount; u++)
    {
        if (!visited[u])
        {
            if (isCyclicUtil(u, visited, -1))
            {
                return true;
            }
        }
    }

    return false;
}
```

Create the **isCyclicUtil()** helper method. It accepts vertex, bool array of visited vertices and a parent vertex:

```
private bool isCyclicUtil(int vertex, bool[] visited, int parent)
{
}
```

In the method, mark the **current vertex** as **visited**. Then, loop through the **adjacent vertices** and check if the vertex has been visited. If an adjacent is **not visited**, then **recur** for that adjacent. If an adjacent is **visited** and **not a parent** of current vertex, then there is a **cycle**. These conditions should return a bool whether there is a cycle or not. The method should look like this:

```
private bool isCyclicUtil(int vertex, bool[] visited, int parent)
{
    visited[vertex] = true;

    foreach (int i in adjacents[vertex])
    {
        if (!visited[i])
        {
            if (isCyclicUtil(i, visited, vertex))
            {
                return true;
            }
        }
        else if (i != parent)
        {
            return true;
        }
    }

    return false;
}
```

Test Code

First, you should create the graph, using the **ReadGraph()** method. Then, check if the graph contains a cycle, using the **isCyclic()** method. If the method returns **true**, print **"Graph contains cycle"** on the console. If it returns **false**, print **"Graph doesn't contain cycle"**.

Press **[Ctrl] + [F5]** to run the program and **test** if it works properly with the examples above.

3. Shortest Path

You are given an **undirected graph**. Write program to find the **shortest path length between two given nodes**. The shortest path length is the minimum number of **edges** between the nodes. Use the **DFS** algorithm.

Input

You need to **read the graph** from the console with the **start** and **end vertices** for the **path**:

- On the first line, you will receive **N**- number of **vertices**
- On the second line, read **M**- number of undirected **edges** between vertices
- On the next **M** lines, read the two vertices, connected by an edge in the format "**{firstVertex} {secondVertex}**"
- After that, on the next line, read the **startVertex** of the path
- On the last line, read the **endVertex** of the path

Find the **shortest path length** (number of edges) between the **startVertex** and the **endVertex**.

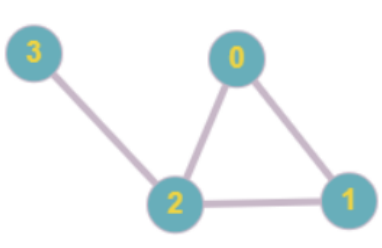
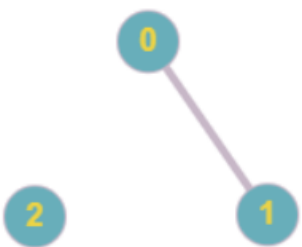
Output

On the first line, print "**Shortest path length from {startVertex} to {endVertex}:**" on the console.

On the next line:

- If you find a **path**, print "**Path found. Length: {shortest path length}**".
- If there is **no path**, print "**No path exists**".

Examples

Input	Graph	Expected Output	Explanation
4 4 0 1 0 2 1 2 2 3 0 3		Shortest path length from 0 to 3: Path found. Length: 2	There are 4 vertices, connected by 4 edges. The path starts from vertex 0 and ends in vertex 3 . The shortest path is 0-2-3 and its length is 2 edges.
3 1 0 1 0 2		Shortest path length from 0 to 2: No path exists	There is no path between vertex 0 and vertex 2 .

Hints

You can use this article (<https://www.geeksforgeeks.org/find-paths-given-source-destination>), in which it is shown how to **print all paths** from a given source vertex to a destination one. Look at **explanations** and **modify the code** to work for the given problem.