

Lab: Graph Theory, Traversal, and Shortest Paths

This document defines the lab for "[Algorithms – Fundamentals \(C#\)](#)" course @ Software University.

Please submit your solutions (source code) of all below-described problems in [Judge](#).

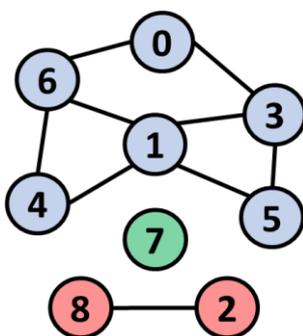
1. Connected Components

The first part of this lab aims to implement the **DFS algorithm** (Depth-First-Search) to **traverse a graph** and find its **connected components** (nodes connected 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
0	(empty graph)	

Hints

DFS Algorithm

First, create a bool array that will be with the size of your graph:

```
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);
    }
}
```

Test

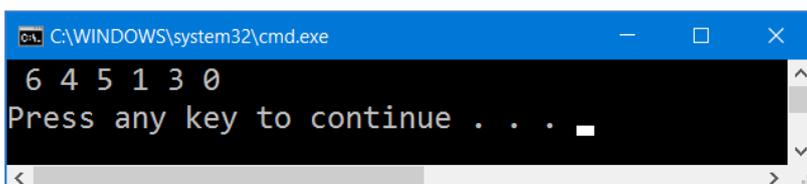
Invoke the **DFS()** method starting from node 0. It should print the connected component, holding the node 0:

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

    DFS(0);

    Console.WriteLine();
}
```

Run the code above by **[Ctrl + F5]**. It should print the first connected component in the graph, holding the node 0:



```
C:\WINDOWS\system32\cmd.exe
6 4 5 1 3 0
Press any key to continue . . .
```

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):

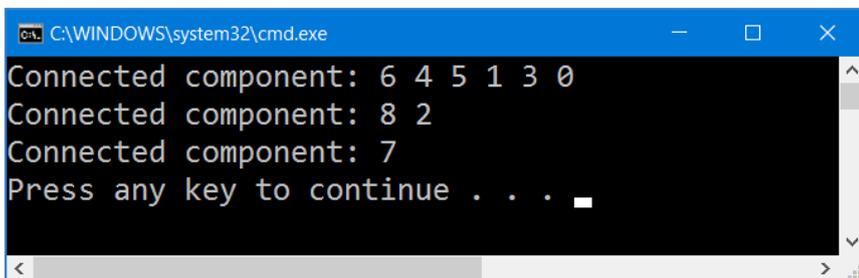
```
private static void FindGraphConnectedComponents()
{
    visited = new bool[graph.Length];

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

Now let's test the above code. Just call it from the main method:

```
public static void Main()
{
    FindGraphConnectedComponents();
}
```

The output is as expected. It prints all connected components in the graph:



```
C:\WINDOWS\system32\cmd.exe
Connected component: 6 4 5 1 3 0
Connected component: 8 2
Connected component: 7
Press any key to continue . . .
```

Read Input

Let's implement the data entry logic (read the 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(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries)
            .Select(int.Parse).ToList();
    }

    return graph;
}
```

Modify the main method to **read the graph from the console** instead of using the hard-coded graph:

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

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

2. Source Removal Topological Sorting

We're given a **directed graph** which means that if node A is connected to node B and the vertex is directed from A to B, we can move from A to B, but not the other way around, i.e. we have a one-way street. We'll call A "**source**" or "**predecessor**" and B – "**child**".

Let's consider the tasks a SoftUni student needs to perform during an exam – "Read description", "Receive input", "Print output", etc.

Some of the tasks **depend on other tasks** – we cannot print the output of a problem before we get the input. If all such tasks are nodes in a graph, a directed vertex will represent dependency between two tasks, e.g. if A -> B (A is connected to B and the direction is from A to B), this means B cannot be performed before completing A first. Having all tasks as nodes and the relationships between them as vertices, we can **order the tasks using topological sorting**.

After the sorting procedure, we'll obtain a list showing all tasks **in the order in which they should be performed**. Of course, there may be more than one such order, and there usually is, but in general, the tasks which are less dependent on other tasks will appear first in the resulting list.

For this problem, you need to implement topological sorting over a directed graph of strings.

Input

- On the first line, you will receive an integer **n** – nodes count.
- On the next **n** lines, you will receive nodes in the following format: "**{key} -> {children1}, {children2},... {childrenN}**".
 - It is possible some of the keys to not having any children.

Output

- If the sorting is possible then print "**Topological sorting: {sortedKey1}, {sortedKey2}, ...{sortedKeyN}**".
- Otherwise, print "**Invalid topological sorting**".

Example

Input	Picture	Output
-------	---------	--------

<p>6</p> <p>A -> B, C</p> <p>B -> D, E</p> <p>C -> F</p> <p>D -> C, F</p> <p>E -> D</p> <p>F -></p>		<p>Topological sorting: A, B, E, D, C, F</p>
<p>5</p> <p>IDEs -> variables, loops</p> <p>variables -> conditionals, loops, bits</p> <p>conditionals -> loops</p> <p>loops -> bits</p> <p>bits -></p>		<p>Topological sorting: IDEs, variables, conditionals, loops, bits</p>
<p>2</p> <p>A -> B</p> <p>B -> A</p>		<p>Invalid topological sorting</p>

We'll solve this using two different algorithms – source removal and DFS.

Source Removal Algorithm

The source removal algorithm is pretty simple – it **finds the node which isn't dependent on any other node** and **removes it** along with all vertices connected to it.

We **continue removing** each node recursively **until we're done** and all nodes have been removed. If there are nodes in the graph after the algorithm is complete, there are circular dependencies (we will throw an exception).

Compute Predecessors

To **efficiently** remove a node at each step, we need to **know the number of predecessors for each node**. To do this, we will calculate the number of predecessors beforehand.

Create a dictionary to store the predecessor count for each node:

```
private static Dictionary<string, List<string>> graph;
private static Dictionary<string, int> predecessorCount;
```

Compute the predecessor count for each node:

```

private static void GetPredecessorCount()
{
    foreach (var node :KeyValuePair<string,List<...>> in graph)
    {
        var key:string = node.Key;
        var children:List<string> = node.Value;

        if (!predecessorCount.ContainsKey(key))
        {
            predecessorCount.Add(key, 0);
        }

        foreach (var child:string in children)
        {
            if (!predecessorCount.ContainsKey(child))
            {
                predecessorCount.Add(key, 0);
            }
            predecessorCount[child] += 1;
        }
    }
}

```

Remove Independent Nodes

Now that we know how many predecessors each node has, we just need to:

1. Find a node without predecessors and remove it
2. Repeat until we're done

We'll keep the result in a list and start a loop that will stop when there is no independent node:

```

private static void TopologicalSort()
{
    var sorted = new List<string>();
    while (predecessorCount.Count > 0)
    {
        // TODO
    }
}

```

Finding a source can be simplified with LINQ. We just need to check if such a node exists; if not, we break the loop:

```

var node :KeyValuePair<string,int> = predecessorCount
    .FirstOrDefault(predicate: d :KeyValuePair<string,int> => d.Value == 0);

if (node.Key == null)
{
    break;
}

```

Removing a node involves several steps:

1. All its child nodes lose a predecessor -> decrement the count of predecessors for each of the children
2. Remove the node from the graph

3. Add the node to the list of removed nodes

```
// TODO: Decrement predecessor count

sorted.Add(key);
predecessorCount.Remove(key);
```

Finally, print the sorted nodes.

Detect Cycles

If we ended the loop and the **predecessorCount** still has nodes, this means there is a cycle. Just add a check after the while loop and print the proper message if the **predecessorCount** is not empty:

```
if (predecessorCount.Count > 0)
{
    Console.WriteLine("Invalid topological sorting");
}
else
{
    // TODO: Print the sorted nodes
}
```

DFS Topological Sorting

DFS Algorithm

The second algorithm we'll use is **DFS**. You can comment on the method you just implemented and rewrite it to use the same unit tests.

For this one, we'll need the following collections:

```
private static Dictionary<string, List<string>> graph;
private static HashSet<string> visited;
private static HashSet<string> cycles;
```

The DFS topological sort is simple – loop through each node. We create a linked list for all sorted nodes because the DFS will find them in reverse order (we will add nodes in the beginning):

```
private static LinkedList<string> TopSort()
{
    var sorted = new LinkedList<string>();

    foreach (var node in graph.Keys)
    {
        TopSortDFS(node, sorted);
    }

    return sorted;
}
```

The **DFS** method shouldn't do anything if the node is already visited; otherwise, it should mark the node as visited and add it to the list of sorted nodes. It should also do this for its children (if there are any):

```

private static void TopSortDFS(string node, LinkedList<string> sorted)
{
    if (visited.Contains(node))
    {
        return;
    }

    visited.Add(node);

    foreach (var child in graph[node])
    {
        TopSortDFS(child, sorted);
    }

    sorted.AddFirst(node);
}

```

Note that we add the node to the result **after** we traverse its children. This guarantees that the node will be added after the nodes that depend on it.

Add Cycle Detection

How do we know if a node forms a cycle? We can add it to a list of cycle nodes before traversing its children. If we enter a node with the same value, it will be in the **cycles** collection, so we throw an exception. If there are no descendants with the same value then there are no cycles, so once we finish traversing the children, we remove the current node from **cycles**.

Exiting the method with an exception is easy, just check if the current node is in the list of cycle nodes at the very beginning of the **DFS** method then, keep track of the cycle nodes:

```

private static void TopSortDFS(string node, LinkedList<string> sorted)
{
    if (cycles.Contains(node))
    {
        throw new InvalidOperationException();
    }

    if (visited.Contains(node))
    {
        return;
    }

    visited.Add(node);
    cycles.Add(node);

    foreach (var child in graph[node])
    {
        TopSortDFS(child, sorted);
    }

    cycles.Remove(node);
    sorted.AddFirst(node);
}

```

3. Shortest Path

You will be given a graph from the console your task is to find the shortest path and print it back on the console. The first line will be the number of Nodes - **N** the second one the number of Edges - **E**, then on each **E** line the edge in form **{destination} – {source}**. On the last two lines, you will read the start node and the end node.

Print on the first line the **length** of the shortest path and the second the **path itself**, see the examples below.

Example

Input	Output
8 10 1 2 1 4 2 3 4 5 5 8 5 6 5 7 5 8 6 7 7 8 1 7	Shortest path length is: 3 1 4 5 7
11 11 1 5 1 4 5 7 7 8 8 2 2 3 3 4 4 1 6 2 9 10 11 9 6 3	Shortest path length is: 2 6 2 3