# Lab: Graph Algorithms

This document defines the **in-class exercises** assignments for the ["Algorithms" course @ Software University](https://softuni.bg/opencourses/algorithms).

For the following exercises, you are given a Visual Studio solution holding a skeleton + unit tests.

# Part I – 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 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

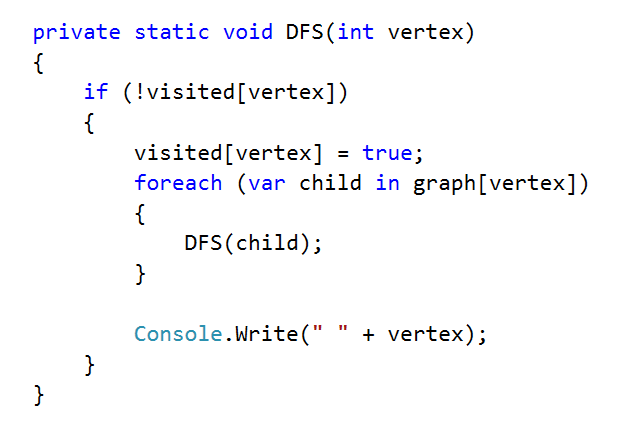
|  |  |  |
| --- | --- | --- |
| **Input** | **Graph** | **Output** |
| 9  3 6  3 4 5 6  8  0 1 5  1 6  1 3  0 1 4  2 |  | Connected component: 6 4 5 1 3 0  Connected component: 8 2  Connected component: 7 |
| 1  0 |  | Connected component: 0 |
| 0 | (empty graph) | Connected component: |
| 7  2 6  1  4  3  1 |  | Connected component: 0  Connected component: 2 6 1  Connected component: 4 3  Connected component: 5 |
| 4  1 2 3  0 1 2 3 3  0 1 3  0 1 1 2 |  | Connected component: 3 2 1 0 |

## DFS Algorithm

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

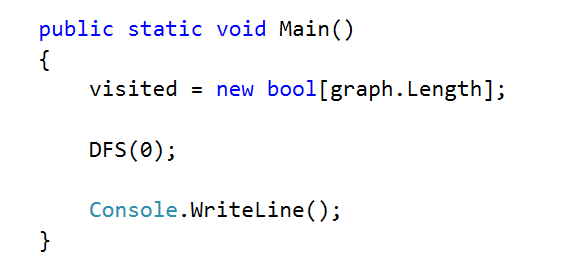


Next, implement the **DFS algorithm** (Depth-First-Search) to traverse all nodes connected to the specified start node:

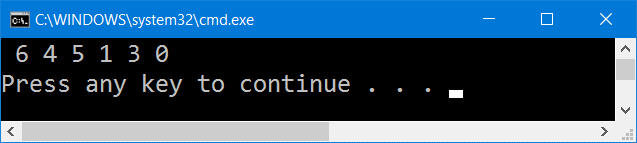


## Test

Now, test the DFS algorithm implementation from the console-based C# project. Invoke the DFS() method starting from node 0. It should print the connected component, holding the node 0:

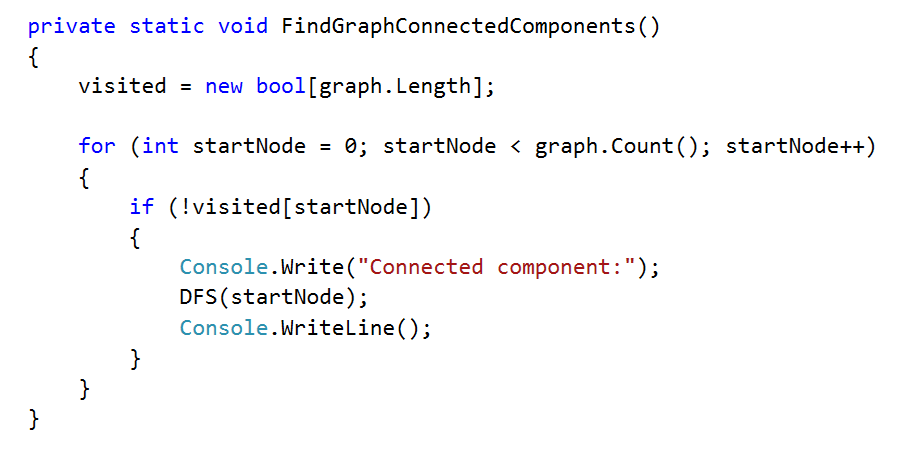


Run the code above by **[Ctrl + F5]**. It should print the first connected component in the graph, holding the node 0:

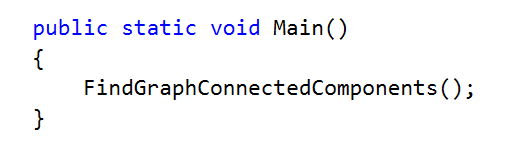


## Find All Components

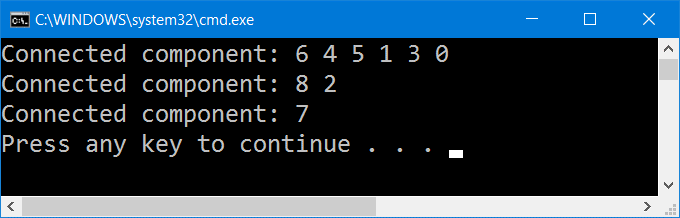
We want to **find all connected components**. We can just run the DFS algorithm for each node taken as a start (which was not visited already):



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

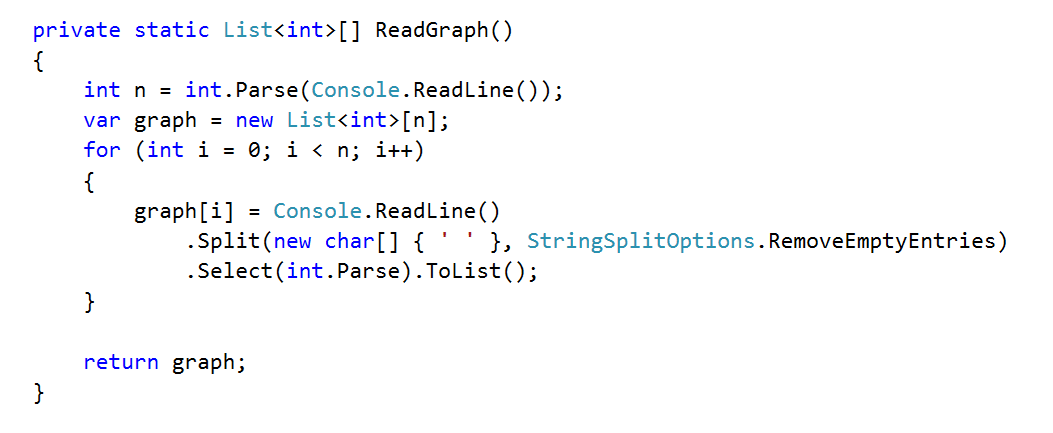


The output is as expected. It prints all connected components in the graph:

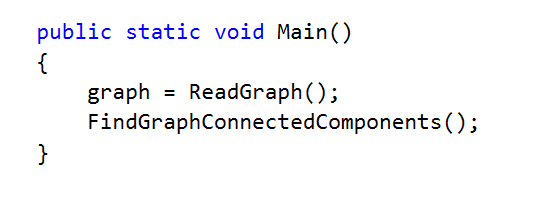


## Read Input

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



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

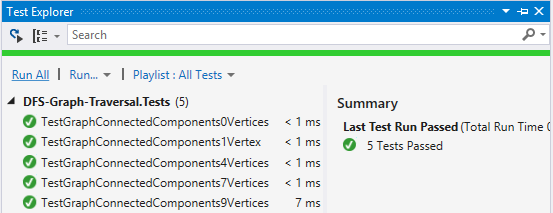


Now test the program. Run it by [Ctrl] + [F5]. Enter a sample graph data and check the output:

|  |  |  |
| --- | --- | --- |
| **Input** | **Graph** | **Expected Output** |
| 7  2 6  1  4  3  1 |  | Connected component: 0  Connected component: 2 6 1  Connected component: 4 3  Connected component: 5  Connected component: 7 |

## Run the Unit Tests

Seems like we solved the graph problem. Let's run the unit tests that come with the program skeleton:



Congratulations! You have implemented the DFS algorithm to find all connected components in a graph.

# Part II – 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.

Clearly, 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.

### Example

|  |  |  |
| --- | --- | --- |
| **Input** | **Picture** | **Output** |
| "A" -> "B", "C"  "B" -> "D", "E"  "C" -> "F"  "D" -> "C", "F"  "E" -> "D"  "F" -> |  | Topological sorting:  A, B, E, D, C, F |
| "IDEs" -> "variables", "loops"  "variables" -> "conditionals", "loops", "bits"  "conditionals" -> "loops"  "loops" -> "bits"  "bits" |  | Topological sorting:  IDEs, variables, conditionals, loops, bits |
| "5" -> "11"  "7" -> "11", "8"  "8" -> "9"  "11" -> "9", "10", "2"  "9" ->  "3" -> "8", "10"  "2" ->  "10" |  | Topological sorting: 3, 7, 8, 5, 11, 2, 10, 9 |

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

## Source Removal Algorithm

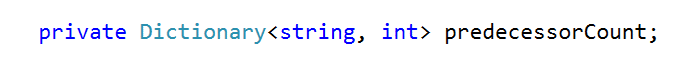
The source removal algorithm is pretty simple – if **finds the node which isn’t dependent on any other node** and **removes it** along with all vertices connected to it.

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 algorithms is complete, there are circular dependencies (we will throw an exception).

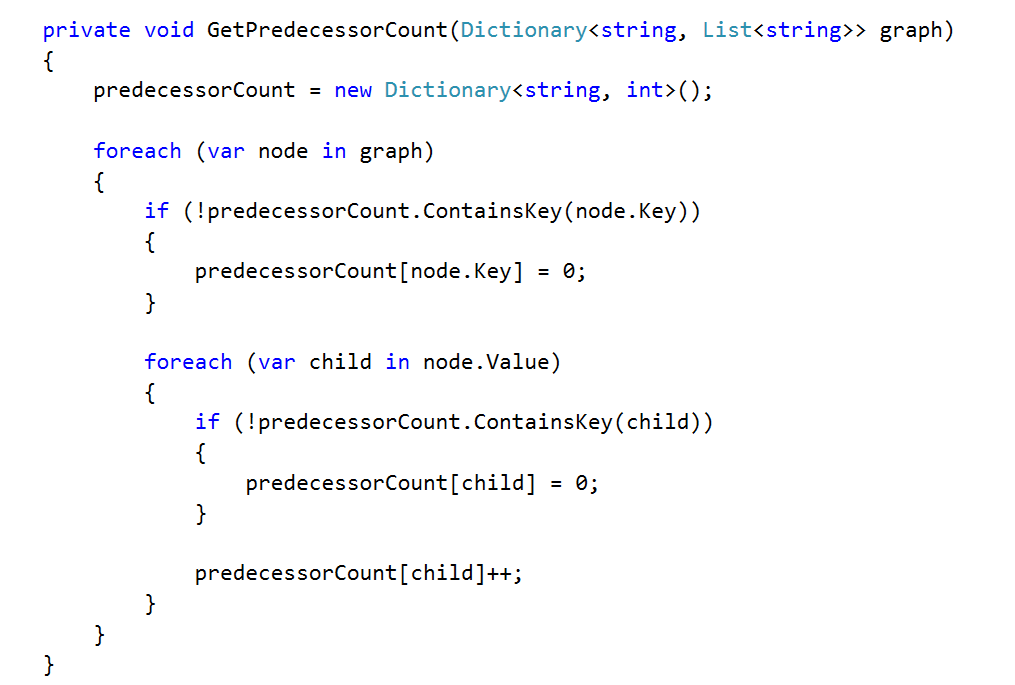
## Compute Predecessors

In order 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:



Compute the predecessor count for each node:

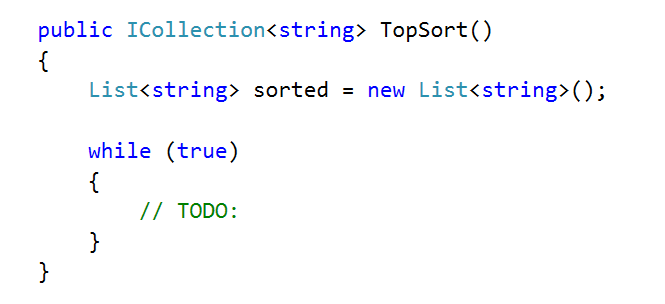


## Remove Independent Nodes

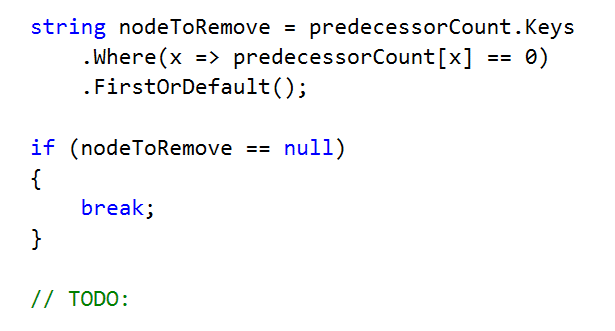
Now that we know how many predecessors each node has, we just need to:

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 which will stop when there is no independent node:

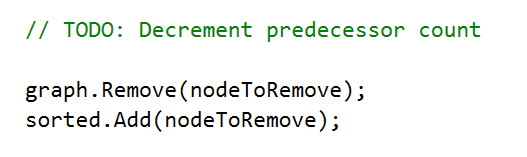


Finding a source can be simplified with LINQ. We just need to check if such a node exists; if not, we break the loop:



Removing a node involves several steps:

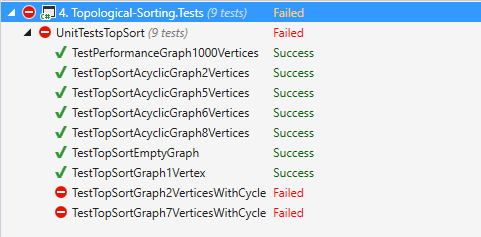
1. All its child nodes lose a predecessor -> decrement the count of predecessors for each of the children
2. Remove the node from the graph
3. Add the node to the list of removed nodes



Finally, return the list of removed nodes.

## Test

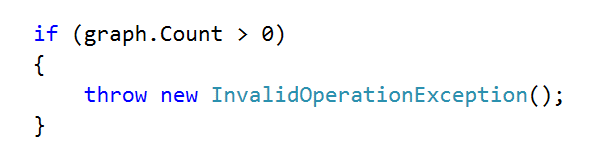
Run the unit tests. It seems we have a problem:



The last unit tests include graphs with cycles in them. We need to modify our algorithm to take care of cycles.

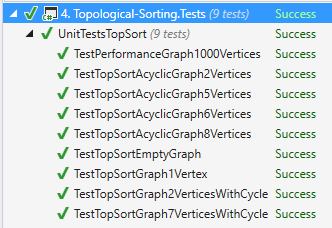
## Detect Cycles

If we ended the loop and the graph still has nodes, this means there is a cycle. Just add a check after the while loop and throw the proper exception if the graph is not empty:



## Test Cycle Detection

Run the unit tests again. This time they should pass:

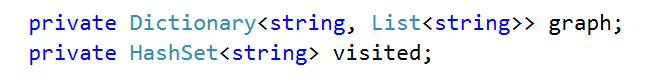


# Part III – DFS Topological Sorting

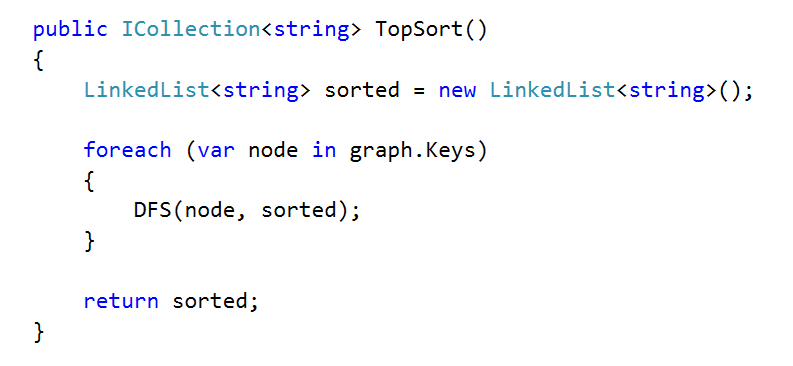
## DFS Algorithm

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

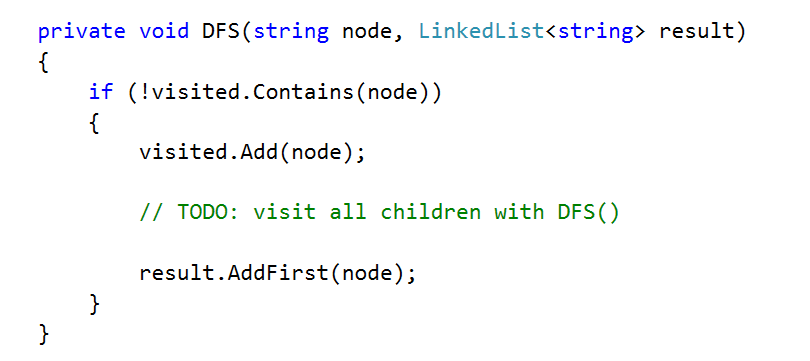
For this one, we’ll need collection that will store all visited nodes:



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):



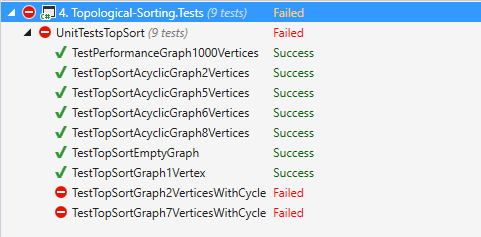
The DFS() method shouldn’t do anything if the node is already visited; otherwise, it should mark the node as visited and add it to the list of sorted nodes. It should also do this for its children (if there are any):



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.

## Test

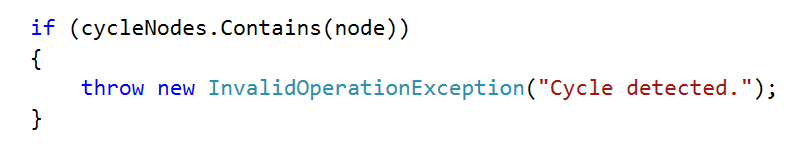
Run the unit tests. Once again, we have problems with cycles:



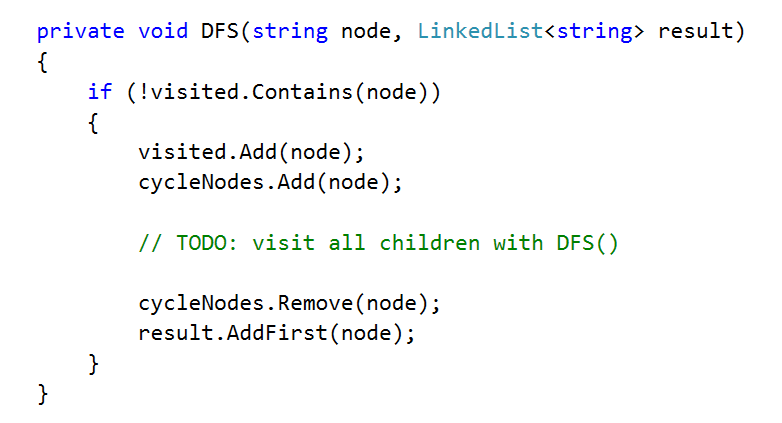
## Add Cycle Detection

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 cycleNodes 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 cycleNodes.

Obviously, we’ll need a new collection to hold the cycle nodes, e.g. a HashSet<string>. 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 TopSortDFS() method:

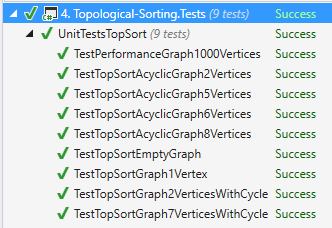


Then, keep track of the cycle nodes:



## Test Cycle Detection

Re-run the unit tests. This time they should all pass:



You have implemented topological sorting using two different algorithms!