# Lab: Graphs Strongly Connected Components, Max Flow

This document defines the lab for the ["Algorithms – Advanced (C#)" course @ Software University](https://softuni.bg/trainings/3810/algorithms-advanced-with-c-sharp-september-2022).

Please submit your solutions (source code) to all below-described problems in [Judge](https://judge.softuni.org/Contests/2579/Graphs-Strongly-Connected-Components-Max-Flow-Lab).

## Strongly Connected Components (SCC)

A strongly connected component is any subgraph in a graph that fulfills the prerequisite that for any two vertices **u** and **v** in the subgraph, there exists a path from **u** to **v**.

For example, you want to check if all towns on a map are **reachable** using the current **road network**. The towns are represented as vertexes, and the roads are represented as edges.

To find the groups in which all towns can reach one another, we need an algorithm for finding **Strongly Connected Components (SCC)**.

We can use the [Kosaraju-Sharir](https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm) algorithm.

### Input

* On the first line, you will receive an integer – n – number of nodes starting from 0.
* On the second line, you will receive an integer – l – number of lines.
* On the next l lines, you will receive a node and its children in the following format: "{node}, {children1}, … {childrenN}".

### Output

* First, print "Strongly Connected Components:" on a single line.
* After that, print all strongly connected components in **topological order** in the following format: "{{node1}, {node2},… {nodeN}}".
  + Nodes in each component should be printed in **topological order** too.

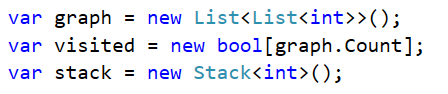
### Example

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Graph** |
| 14  13  0, 1, 11, 13  1, 6  2, 0  3, 4  4, 3, 6  5, 13  6, 0, 11  7, 12  8, 6, 11  9, 0  10, 4, 6, 10  12, 7  13, 2, 9 | Strongly Connected Components:  {10}  {8}  {7, 12}  {5}  {3, 4}  {0, 9, 6, 1, 2, 13}  {11} |  |

### Solution

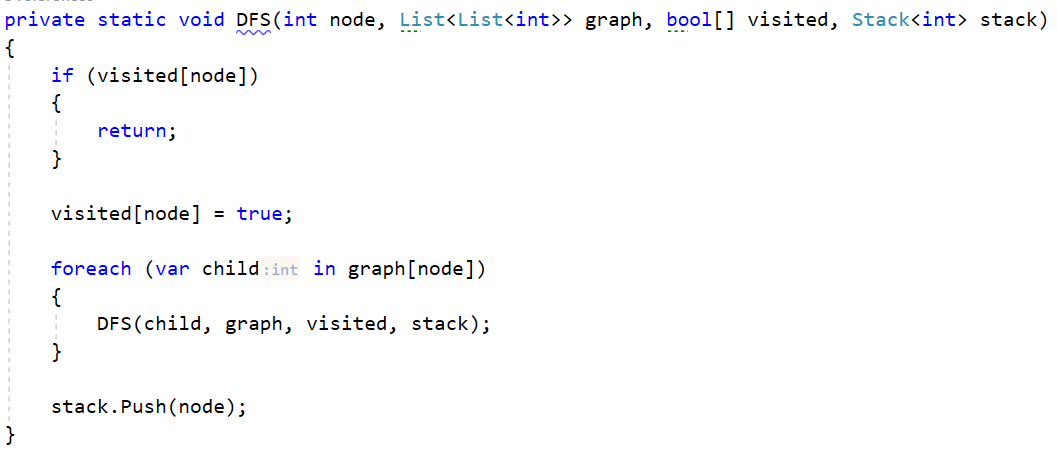
#### Initialize Variables

We should start with the following variables:



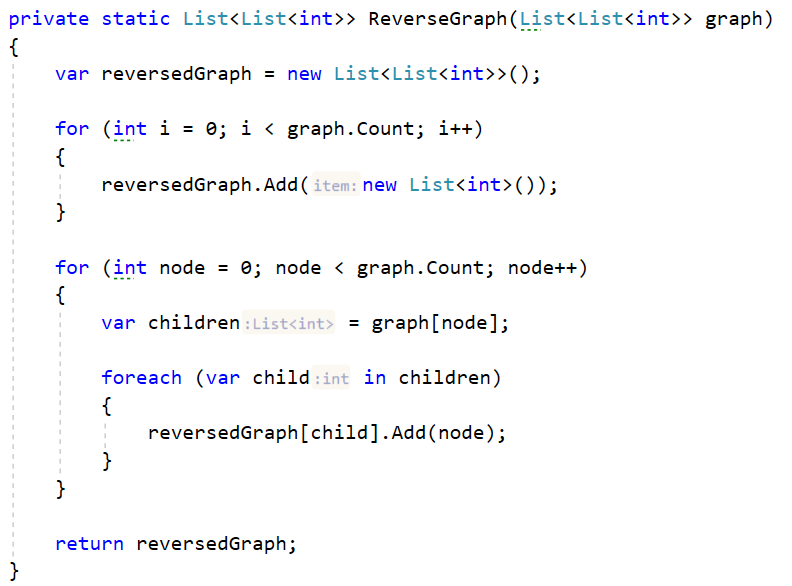
#### DFS

The **Kosaraju-Sharir** algorithm works by first finding all components of the graph. This can be done by using DFS, each time a DFS finishes, all visited elements represent a connected component. For every component, we also want the order in which we visited the vertexes, we can store this information by using a **Stack**, every time the DFS returns from a vertex (all its children have been visited), we **add the vertex to the Stack**. Since the first vertex that will be added to the Stack is the deepest one, when we're finished on top of the Stack will be the vertex we started from.



#### Build Reverse Graph

After knowing all connected components we have all connected components, we need to build the reverse graph. Building it is not hard we loop through the adjacency list, and for each connection **u -> v,** we instead add a connection **v -> u** in the reversed graph:



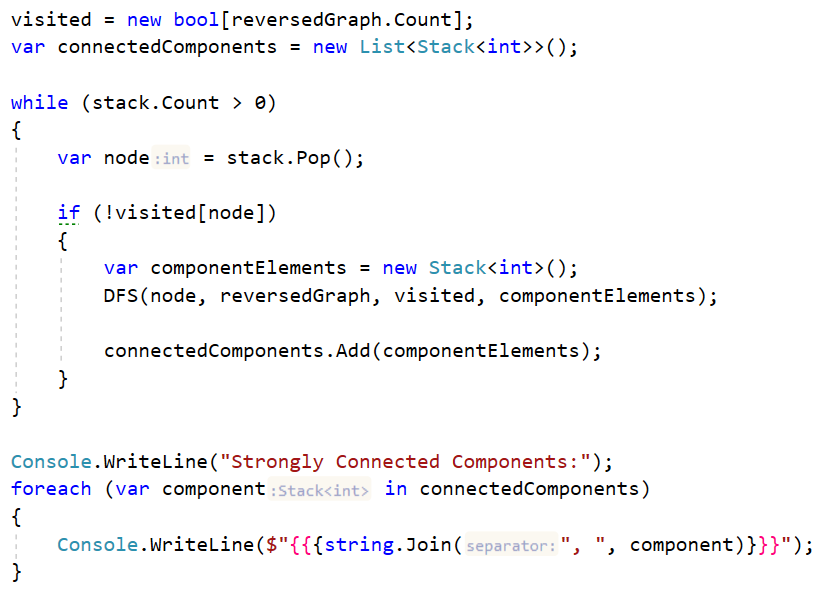
#### Extract Strongly Connected Components

After we have the reversed graph, we can find the strongly connected components.

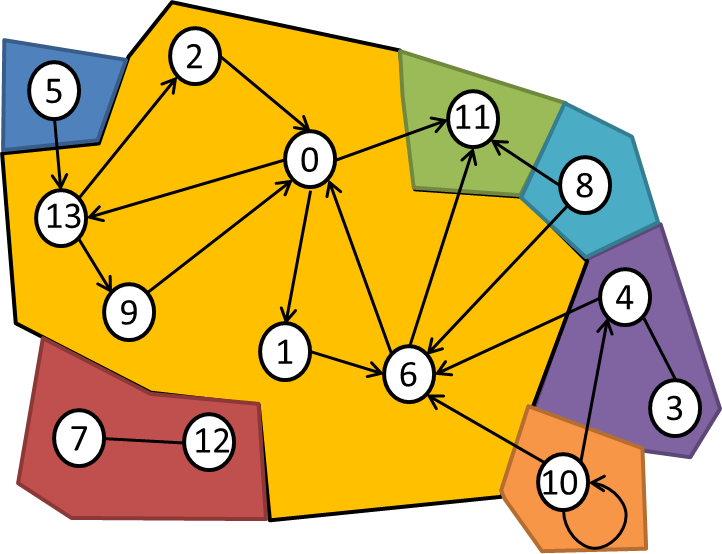
The conjecture is the following, if a vertex is part of an **SCC** (strongly connected component), then there is a path from that vertex to any other in the SCC and from any other in the SCC to that vertex, if that is true then reversing the edges should not change this (if there is a path from **u -> v** and from **v -> u**, then reversing the edges would mean there is a path from **v -> u** and **u -> v**).

Starting from the top of the Stack, we pop an element and start a DFS in the reversed graph.

Every time the ReverseDFS function finishes, all nodes we visited represent a strongly connected component.



And this is the result visually:



## Max Flow algorithm - Edmonds-Karp

Max flow algorithms are important because many problems can be modified and then represented as a max flow problem. In this problem, we'll try to implement the [Edmonds-Karp](https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm) algorithm.

### Input

* On the first line, you will receive an integer – n – number of nodes.
* On the next n lines, you will receive n count elements separated by ", ".
  + Each element will represent the initial capacity for the given from and to indices.
    - Assume from the index is the line index and to index is the element index.
* On the next line, you will receive an integer – source – starting node.
* On the next line, you will receive an integer – destination – end node.

### Output

* Print the max flow in the following format: "Max flow = {maxFlow}".

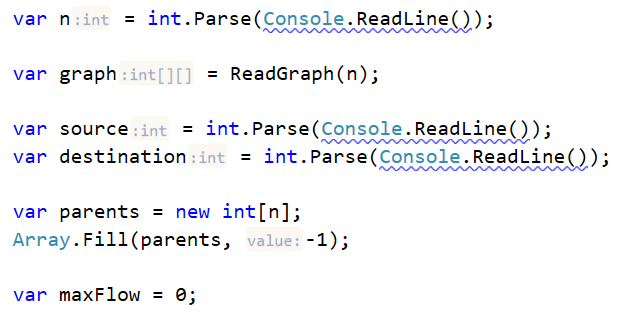
### **Example**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Graph** |
| 6  0, 10, 10, 0, 0, 0  0, 0, 2, 4, 8, 0  0, 0, 0, 0, 9, 0  0, 0, 0, 0, 0, 10  0, 0, 0, 6, 0, 10  0, 0, 0, 0, 0, 0  0  5 | Max flow = 19 | Graph: |

### Solution

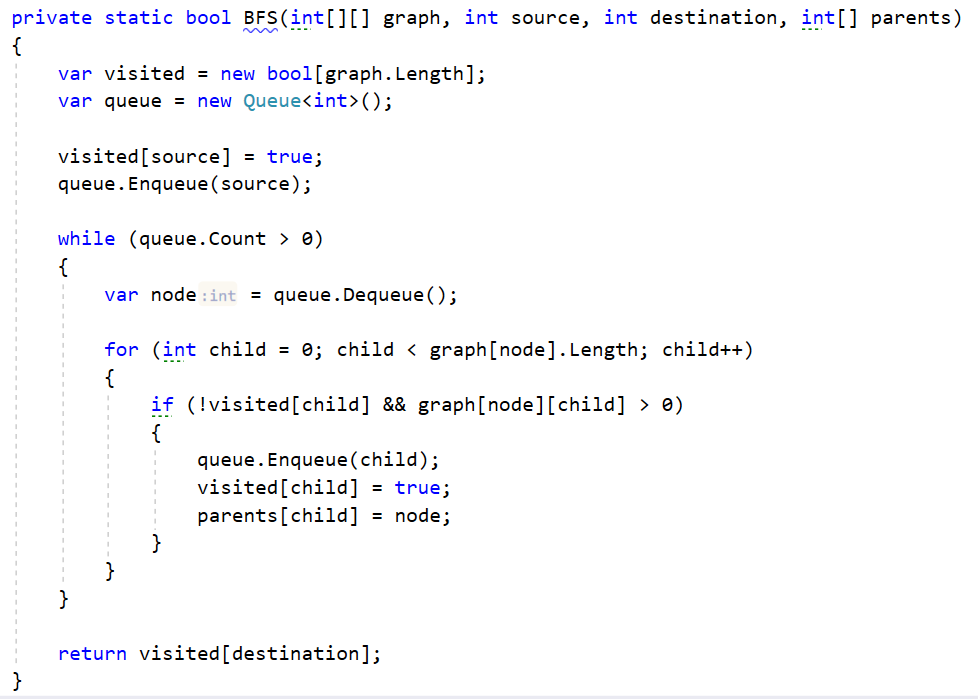
#### Initialize Variables

Max flow algorithms rely on **finding a path** from the **start vertex (source)** to the **end vertex (sink)**. Such a path is called an augmenting path. After finding such a path, we usually have to trace it back to the start, so we need to keep a collection to see where we came from. We'll create an **array for keeping track of the parents of vertexes**, to know if we have set a parent for a vertex or not, we'll initialize the parents' array with starting value **-1**:



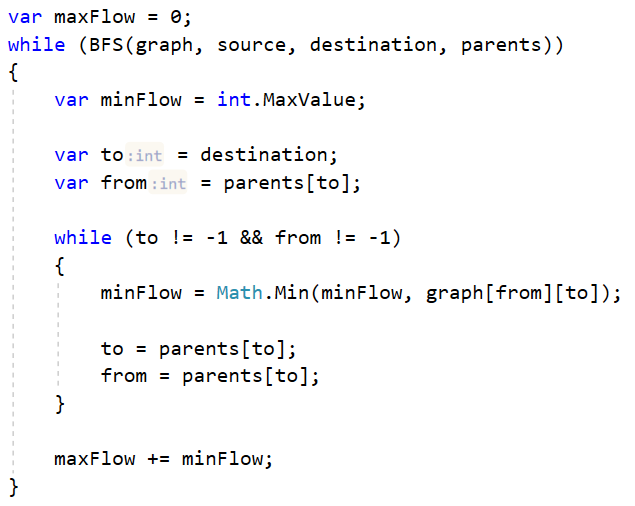
#### Find Augmenting Paths

The key point in Max flow algorithms is finding an augmenting path. The **Edmonds-Karp** algorithm is an implementation of the **Ford-Fulkerson** algorithm where we find the augmenting path using BFS:



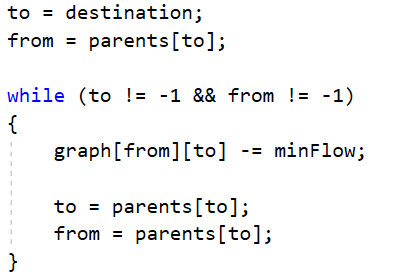
#### Reconstruct Path

While the BFS algorithm keeps finding augmenting paths, we'll traverse the path starting from the end/sink back while keeping track of the **smallest edge capacity** we've passed. We keep track of the smallest capacity in our path because the most amount of flow we can pass through the path is at most the amount we can pass through the smallest edge.



#### Modify the Graph

After finding the most flow we can have through the augmenting path, we **add it to the max flow,** and then we modify the graph to **mark that we have filled the found augmenting path**. We subtract from each edge in the augmenting path's capacity the amount of flow we've run through the path:



With this, we should be done the only thing left is to print the max flow.

## Articulation Points

Sometimes we need to find articulation points (cut vertices) in a certain network – **points that, when removed, will split the graph into separated components**.

In situations such as networks for ISPs (Internet Service Providers), electricity, or water, it is very important to have redundancy in the networks. If between every vertex **u** and vertex **v** in a network, there existed at least 2 different paths (paths that have no vertexes in common aside from the beginning and the end vertex), such a network would be called **biconnected**. However, if there existed some point **x** (different from **u** and **v**) which is always a part of any path between **u** and **v**, such a point would be an articulation point.

Let's implement the [Hopcroft-Tarjan](https://en.wikipedia.org/wiki/Biconnected_component) algorithm for finding the articulation points in the following graph:

### Input

* On the first line, you will receive an integer – n – number of nodes starting from 0.
* On the second line, you will receive an integer – l – number of lines.
* On the next l lines, you will receive a node and its children in the following format: "{node}, {children1}, … {childrenN}".

### Output

* Print all articulation points in the following format: "Articulation points: {articulationPoint1}, …, { articulationPointN}".

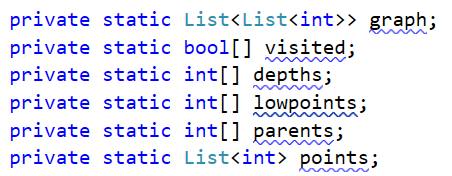
### Example

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Graph** |
| 12  12  0, 1, 2, 6, 7, 9  1, 0, 6  2, 0, 7  3, 4  4, 3, 6, 10  5, 7  6, 0, 1, 4, 8, 10, 11  7, 0, 2, 5, 9  8, 6, 11  9, 0, 7  10, 4, 6  11, 6, 8 | Articulation points: 4, 6, 7, 0 |  |

### Solution

#### Initialize Variables

First, we'll initialize the collections we need, we'll need something to **store the graph** we'll work with, and we can also initialize a collection to **keep the articulation points**. We'll also need collections to **hold the depths**, **low points,** and the **parents of the vertices** and a collection to **keep visited vertices**:

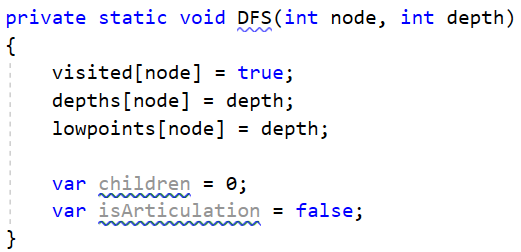


#### Create the Main Recursive Function

Our algorithm works based on DFS, we need to **create the recursive function**, as parameters, we'll pass the index of the **starting node** and the **starting depth**.

At the beginning of the function, we'll **mark the current node as visited** and **initialize its depth** and **low point** to the depth parameter.

Locally in the stack frame, we'll also **keep track of the number of children of the vertex,** and to simplify our checks, we'll also use a variable to keep track of the current point is an articulation point.



#### Implement the Main Loop

As we mentioned, the algorithm is based on DFS, so we'll implement the main loop in a DFS where we traverse all children of a vertex. If a child isn't already visited, we **mark its parent** as the current vertex and **recursively call the function for the child with an increased depth**.

After returning from the recursion, we **increment the child count** and **compare the low point of the child compared to the depth of the current vertex**. If the child's low point is equal to or bigger than the current depth, we know it hasn't found a different path to a previous node, so the current vertex must be an articulation point. After the check, we **set the low point for the current vertex** to be the smaller between itself and the child's low point.

In case the child was visited but is not our direct parent (so we avoid the situation where we go to a child and check back to where we came from), we **set the low point of the current vertex** to be the smaller of itself and the depth of the visited child.

After we have visited all children of a vertex, we can conclude whether it's an articulation point or not. If the current vertex is the starting vertex and it has at least 2 children (each in a different component), it's an articulation point, alternatively if it's not the starting node and the **isArticulation** boolean is true, it's also an articulation point.

