# Lab: Advanced 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 "**Advanced-Graph-Algorithms-II-Lab**" holding portions of the source code + unit tests. You can download it from the course’s page.

# Part I – Strongly Connected Components (SCC) – Kosaraju- Sharir Algorithm

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 actually **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)**.

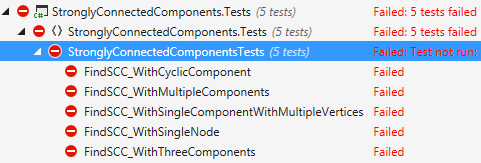
We can use the [Kosaraju-Sharir](https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm) algorithm.

### Example

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

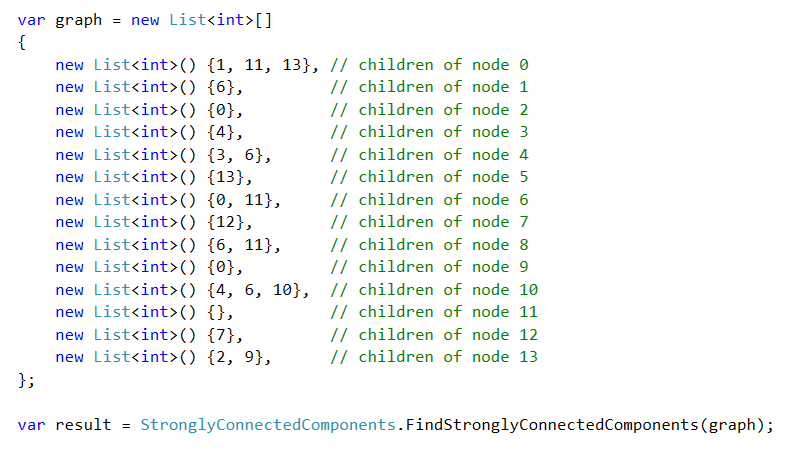
## Run the Unit Tests

Run the unit tests in the **StronglyConnectedComponents.Tests** project. Since we haven’t written the algorithm yet, so they should all fail:

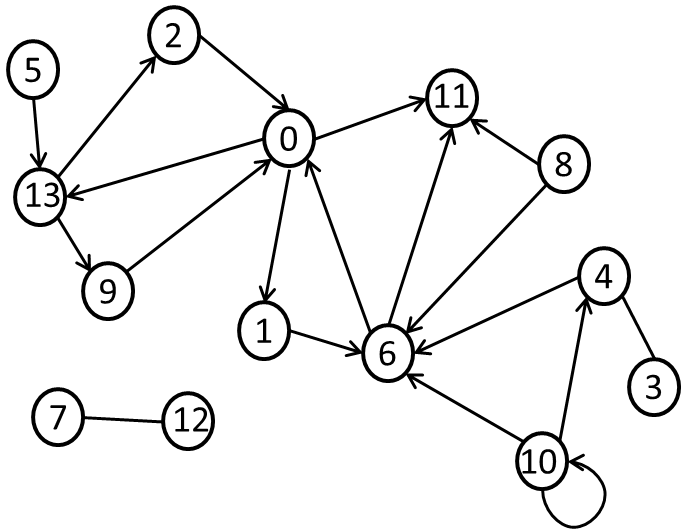


## Start with a Hardcoded Example

In the **StronglyConnectedComponentsMain** class you’re provided with a test case – a graph with 14 nodes represented as an adjacency list. Your task is to complete the FindStronglyConnectedComponents() method in the **StronglyConnectedComponents** class, which receives the graph and returns the strongly connected components as a **List<List<int>>**, where each inner list represents a **strongly connected component** (scc).

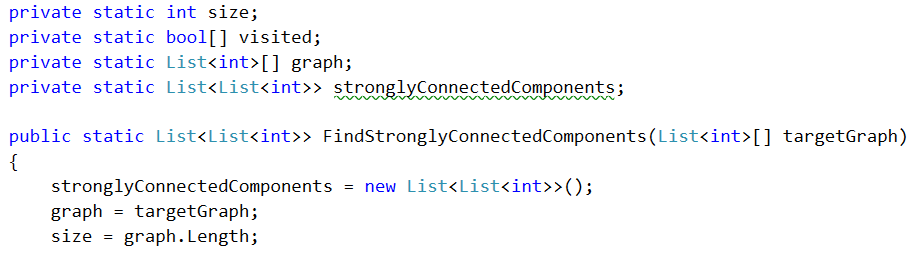


This is how this graph looks like:



## Initialize Variables

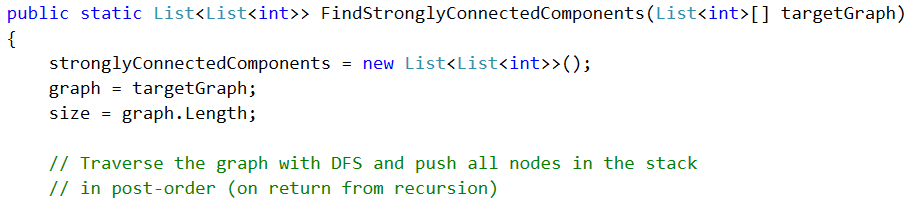
We should store the passed in graph and **initialize the collection** that will keep the results.

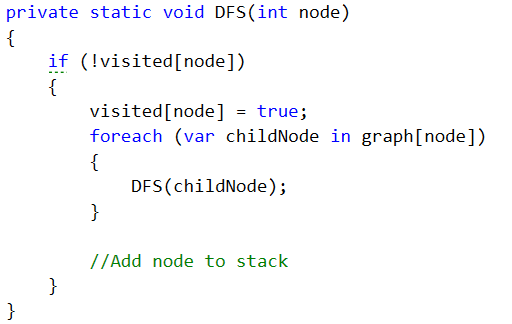


## DFS

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.

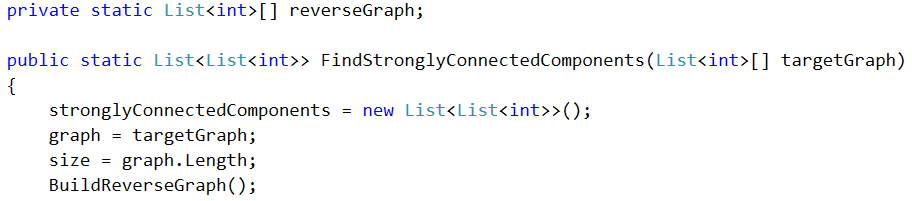


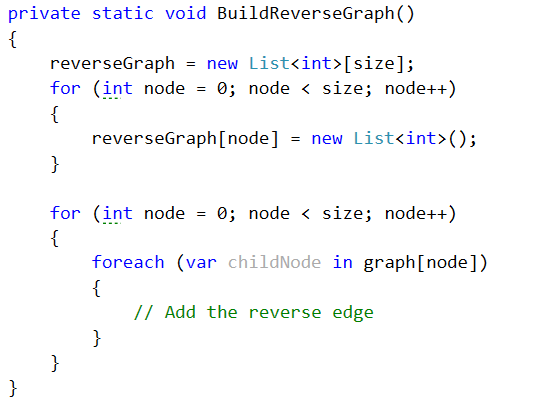




## Build Reverse Graph

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:



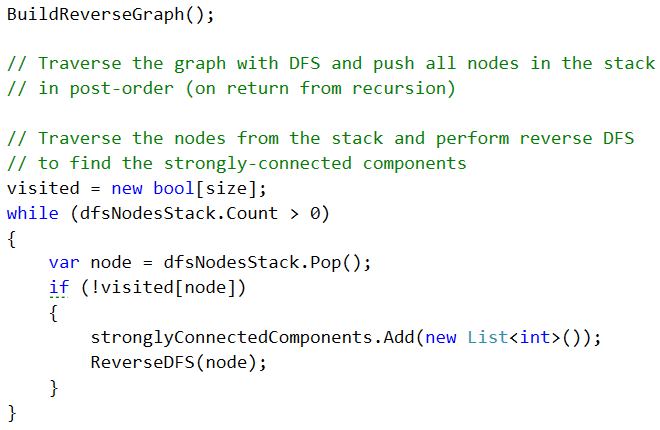


## Reverse DFS

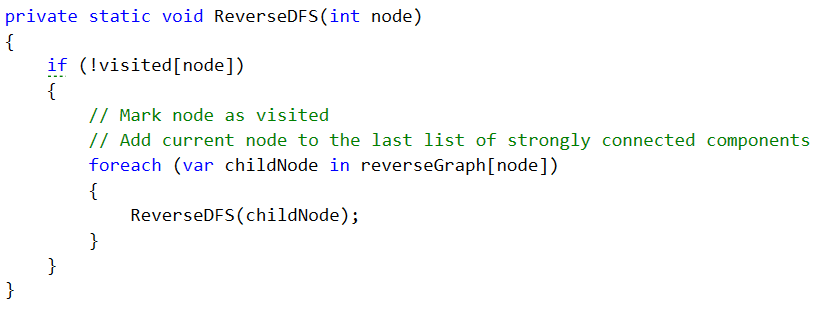
After we have the reversed graph, we can find the strongly connected components.

The conjecture is the following, if a vertex is part of a **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 than 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 from **u -> v**).

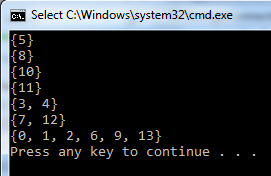
Starting from the top of the Stack we pop an element and start a DFS in the reversed graph:



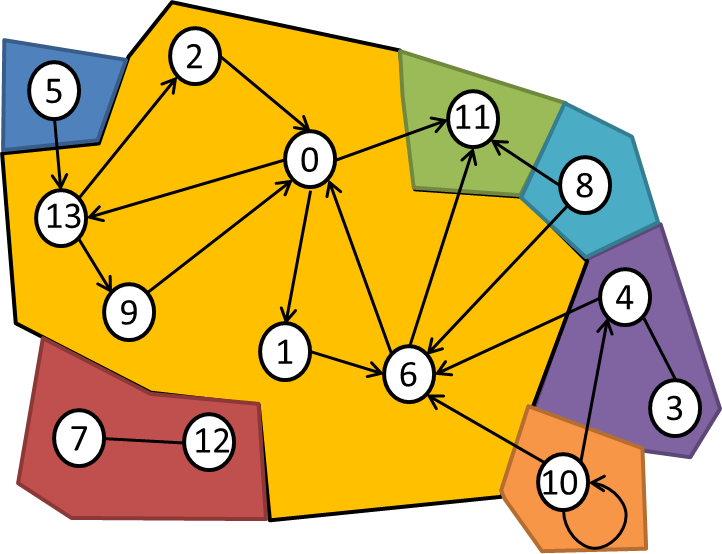
Every time the ReverseDFS function finishes all nodes we visited represent a strongly connected component.



Once you’re done, run the program. It should produce the following result:

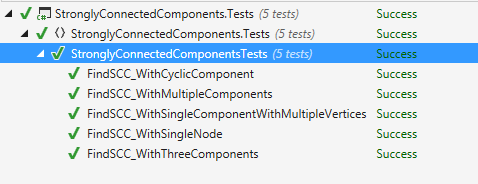


And this is the result visually:



## Run the Unit Tests Again

This time all tests should pass:



You’ve successfully implemented the **Kosaraju-Sharir** algorithm!

# Part II – 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**.

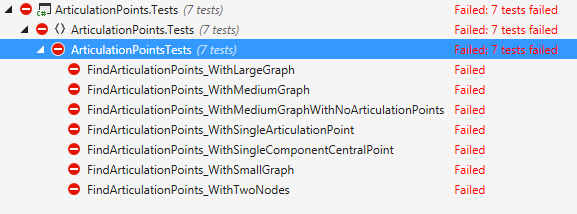
In situations such as networks for ISPs (Internet Service Providers), electricity or water it is very important to have redundancy in the networks. If between every vertex **u** and vertex **v** in a network there existed at least 2 different paths (paths that have no vertexes in common aside from the 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:

|  |  |
| --- | --- |
| **Graph** | **Output** |
|  | Articulation points: 4, 6, 7, 0 |

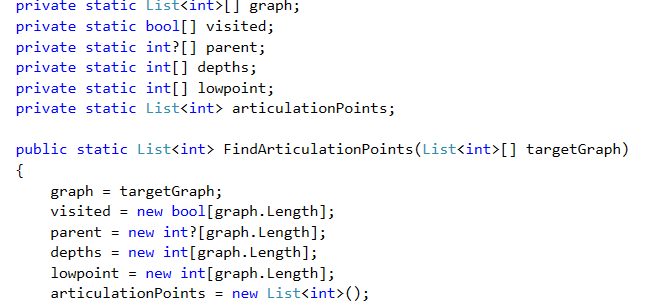
## Run the Unit Tests to Make Sure They Initially Fail

The algorithm is still not implemented, so all unit tests should fail:



## Initialize Collections

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**, **lowpoints**, **parents of the vertices** and a collection to **keep visited vertices**:

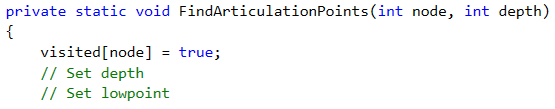


## Create the Main Recursive Function

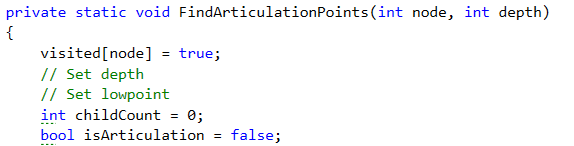
Our algorithm works on the basis of DFS, we need to **create the recursive function**, as parameters we’ll pass the index of the **starting node** and the **starting depth**:



In the beginning of the function we’ll **mark the current node as visited** and **initialize its depth** and **lowpoint** to the depth parameter.

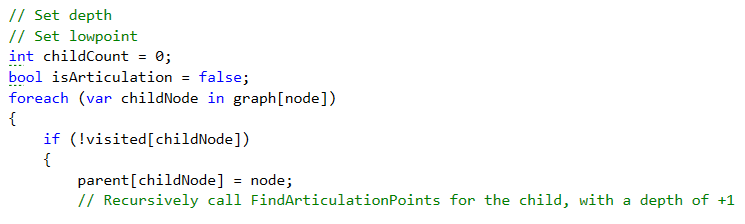


Locally in the stack frame we’ll also **keep track of the number of children of the vertex** and to simplify our checks we’ll also use a variable to keep track if the current point is an articulation point.

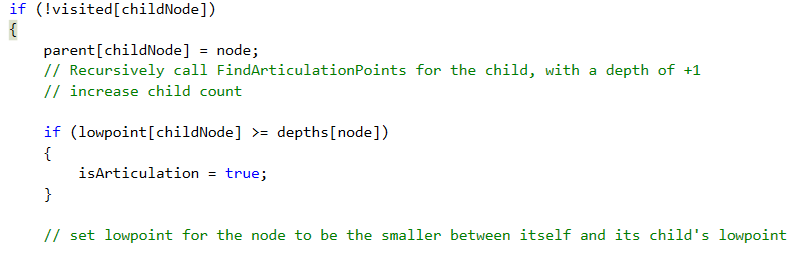


## Implement the Main Loop

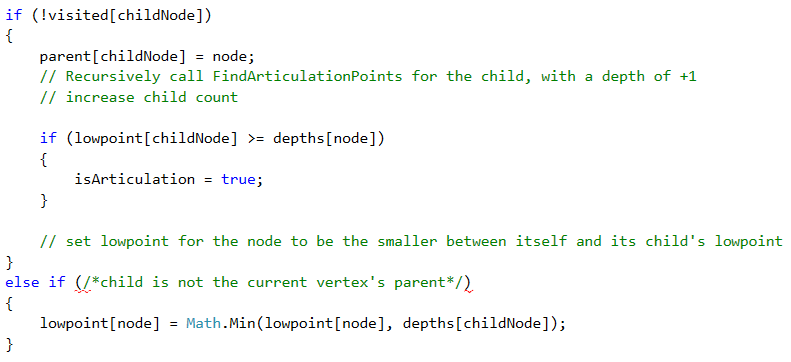
As we mentioned the algorithm is based on DFS, so we’ll implement the main loop in a DFS where we traverse all children of a vertex. If a child isn’t already visited, we **mark its parent** as the current vertex and **recursively call the function for the child with an increased depth**:



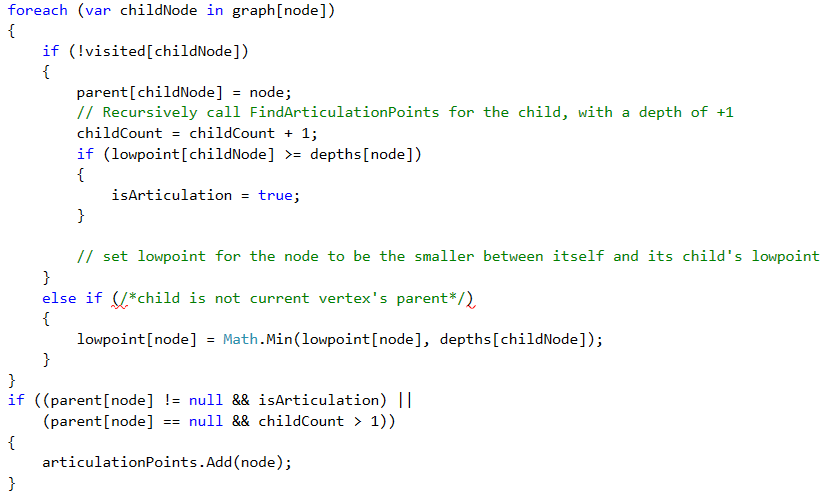
After returning from the recursion, we **increment the child count** and **compare the lowpoint of the child compared to the depth of the current vertex**. If the child’s lowpoint is equal to or bigger than the current depth, we know it hasn’t found a different path to a previous node, so the current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



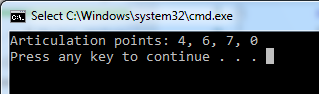
In case the child was visited, but is not our direct parent (so we avoid the situation where we go to a child and check back to where we came from) we **set the lowpoint of the current vertex** to be the smaller of itself and the depth of the visited child:



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.

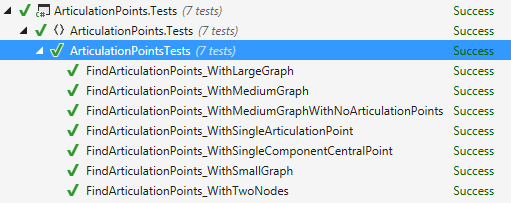


Once you’re done, run the program. It should produce the following result:



## Run the Unit Tests Again

If you’ve completed all tasks correctly, the unit test should now pass:



Congratulations! You have successfully completed the Hopcroft-Tarjan algorithm for finding articulation points!

# Part III – Max Flow algorithm - Edmonds-Karp

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

Example

Find the maximum flow possible in the network starting from **node 0** and ending in **node 5**

|  |  |
| --- | --- |
| **Graph** | **Output** |
|  | Max flow = 19 |

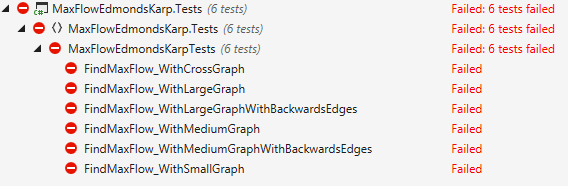
## Provided Assets

You are provided with a test case in the **EdmondsKarp**Main() method. The graph is given as an adjacency matrix where the rows and the columns are the vertexes and the cells represent the capacity of the edge. For example cell **[2, 4]** with value 50 would mean that there is an edge from **node 2** to **node 4** with **capacity 50**, if a cell’s value is 0 then there is no edge between the nodes.

## Run the Unit Tests to Make Sure They Initially Fail

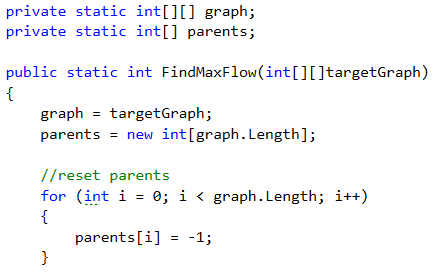
Run the Unit Tests to Make Sure They Initially Fail

The algorithm is still not implemented, so all unit tests should fail:



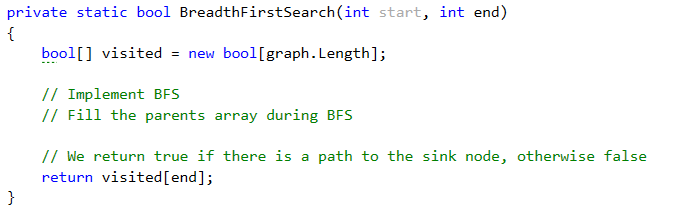
## Initialize Structures

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**, in order to know if we have set a parent for a vertex or not we’ll initialize the parents array with starting value **-1**:



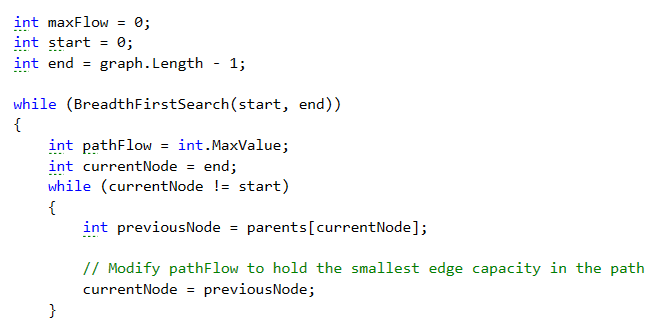
## Find Augmenting Paths

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:



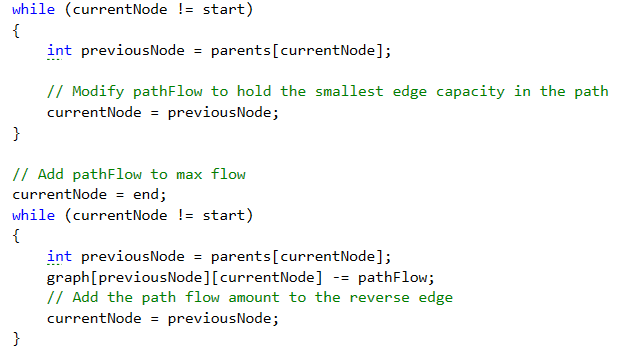
## Reconstruct Path

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

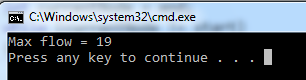
After finding the most flow we can have through the augmenting path, we **add it to the max flow** and then we modify the graph to **mark that we have filled the found augmenting path**. We subtract from each edge in the augmenting path’s capacity the amount of flow we’ve run through the path and increase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



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

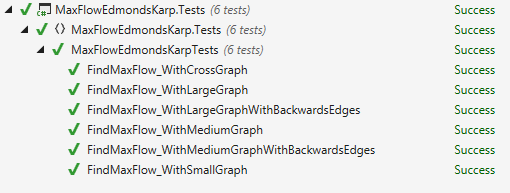


If you’ve done everything correctly, you should get:



## Run the Unit Tests Again

If you’ve completed all tasks correctly, the unit test should now pass:



Congratulations! You’ve implemented the **Edmonds-Karp** algorithm!