Lab: Graph Theory, Traversal and Shortest Paths

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

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

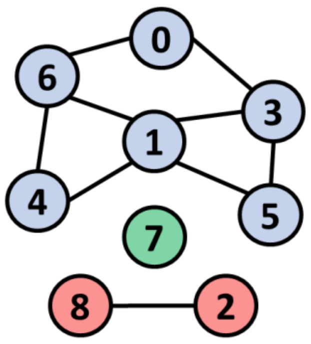
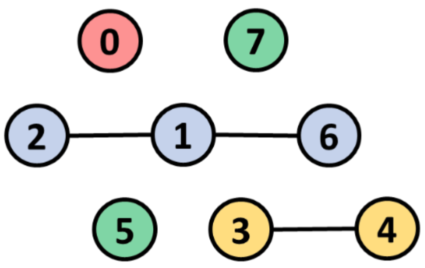
1. Connected Components

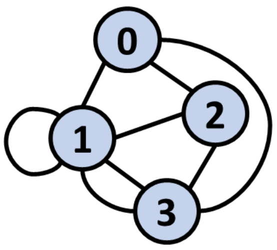
The first part of this lab aims to implement the **DFS algorithm** (Depth-First-Search) to **traverse a graph** and find its **connected components** (nodes connected to each other either directly, or through other nodes). The graph nodes are numbered from **`0`** to **`n-1`**. The graph comes from the 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`		Connected component: 6 4 5 1 3 0
`3 6`		Connected component: 8 2
`3 4 5 6`		Connected component: 7
`8`		
`0 1 5`		
`1 6`		
`1 3`		
`0 1 4`		
`2`		
`1`		Connected component: 0
`0`		
`0`	(empty graph)	Connected component:
`7`		Connected component: 0
		Connected component: 2 6 1
`2 6`		Connected component: 4 3
`1`		Connected component: 5
`4`		
`3`		
`1`		

`4`
`1 2 3`
`0 1 2 3 3`
`0 1 3`
`0 1 1 2`
		Connected component: 3 2 1 0

1. DFS Algorithm

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

```
public static boolean[] visited;
```

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

```
public static void dfs(int vertex) {
    if (!visited[vertex]) {
        visited[vertex] = true;
        for (int child : graph.get(vertex)) {
            dfs(child);
        }
        System.out.println(" " + vertex);
    }
}
```

2. Test

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

```
public static void main(String[] args) throws IOException {
    graph = readGraph();

    visited = new boolean[graph.size()];

    dfs( vertex: 0);
}
```

```
 6 4 5 1 3 0
Process finished with exit code 0
```

3. 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):

```java
public static List<Deque<Integer>> getConnectedComponents(List<List<Integer>> graph) {
    List<Deque<Integer>> components = new ArrayList<>();

    visited = new boolean[graph.size()];

    for (int startNode = 0; startNode < graph.size(); startNode++) {
        if (!visited[startNode]) {
            Deque<Integer> currentComponents = new ArrayDeque<>();
            dfs(startNode, graph, currentComponents);
            components.add(currentComponents);
        }
    }

    return components;
}
```

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

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.

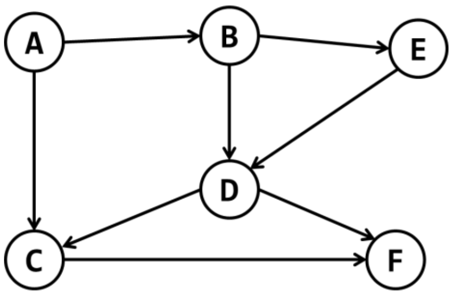
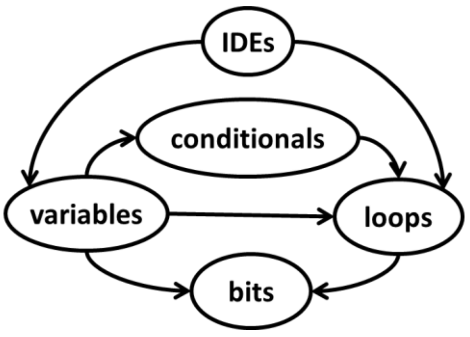
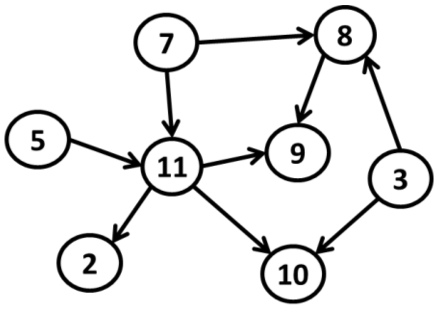
Some of the tasks **depend on other tasks** – we cannot print the output of a problem before we get the input. If all such tasks are nodes in a graph, a directed vertex will represent dependency between two tasks, e.g. if A -> B (A is connected to B and the direction is from A to B), this means B cannot 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.

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.

1. 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).

2. 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 map to store the predecessor count for each node:

Compute the predecessor count for each node:

```java
private static Map<String, Integer> getPredecessorCount(Map<String, List<String>> graph) {
    Map<String, Integer> predecessorCount = new HashMap<>();

    for (Map.Entry<String, List<String>> node : graph.entrySet()) {
        predecessorCount.putIfAbsent(node.getKey(), 0);
        for (String child : node.getValue()) {
            predecessorCount.putIfAbsent(child, 0);
            predecessorCount.put(child, predecessorCount.get(child) + 1);
        }
    }

    return predecessorCount;
}
```

3. 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:

```java
public static List<String> topSort(Map<String, List<String>> graph) {
    List<String> sorted = new ArrayList<>();

    while (true) {
        // TODO
    }
}
```

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

```java
while (true) {
    String nodeToRemove = predecessorCount.entrySet() Set<Map<K, V>.Entry<String, Integer>>
            .stream() Stream<Map<K, V>.Entry<String, Integer>>
            .filter(e -> e.getValue() == 0)
            .map(e -> e.getKey()) Stream<String>
            .findFirst() Optional<String>
            .orElse( other: null);

    if (nodeToRemove == 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

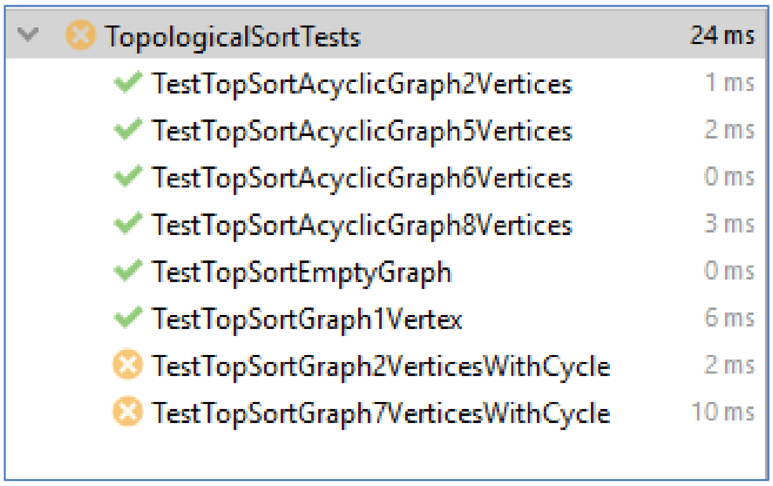
2. Remove the node from the graph
3. Add the node to the list of removed nodes

```
graph.remove(nodeToRemove);
sorted.add(nodeToRemove);
```

Finally, return the list of removed nodes.

4. Test

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



```
TopologicalSortTests                        24 ms
    ✓ TestTopSortAcyclicGraph2Vertices       1 ms
    ✓ TestTopSortAcyclicGraph5Vertices       2 ms
    ✓ TestTopSortAcyclicGraph6Vertices       0 ms
    ✓ TestTopSortAcyclicGraph8Vertices       3 ms
    ✓ TestTopSortEmptyGraph                  0 ms
    ✓ TestTopSortGraph1Vertex                6 ms
    ✗ TestTopSortGraph2VerticesWithCycle     2 ms
    ✗ TestTopSortGraph7VerticesWithCycle    10 ms
```

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

5. 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:

```
if (!graph.isEmpty()) {
    throw new IllegalArgumentException();
}
```

6. Test Cycle Detection

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

3. DFS Topological Sorting

1. 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 a collection that will store all visited nodes:

```java
public static List<String> topSort(Map<String, List<String>> graph) {
    Set<String> visited = new HashSet<>();

}
```

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):

```java
public static List<String> topSort(Map<String, List<String>> graph) {
    Set<String> visited = new HashSet<>();

    List<String> sorted = new ArrayList<>();

    for (String node : graph.keySet()) {
        topSortDfs(sorted, visited, node, graph);
    }

    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):

```java
private static void topSortDfs(List<String> sorted, Set<String> visited, String node, Map<String, List<String>> graph) {
    if (!visited.contains(node)) {
        visited.add(node);

        for (String child : graph.get(node)) {
            topSortDfs(sorted, visited, child, graph);
        }

        sorted.add( index: 0, 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.

2. Test

Run the unit tests. Once again, we have problems with cycles, in addition, there **might** be **some test** results that **differ** in terms of the **order**, think about it we do **different** traversal, so we can get a **different** correct answer.

3. 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 **`Set<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 **DFS** method then, keep track of the cycle nodes:

```java
if (cycles.contains(node)) {
    throw new IllegalArgumentException();
}

if (!visited.contains(node)) {
    visited.add(node);
    cycles.add(node);
    for (String child : graph.get(node)) {
        topSortDfs(sorted, visited, child, graph, cycles);
    }
    cycles.remove(node);
    sorted.add( index: 0, node);
}
```

4. Test Cycle Detection

Re-run the unit tests. This time they should all pass. You have implemented topological sorting using two different algorithms!

4. 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 on the second the **path itself,** see the examples below.

Hint: Try to

Example

Input	Output

`8`
`10`
`1 2`
`1 4`
`2 3`
`4 5`
`5 8`
`5 6`
`5 7`
`5 8`
	`Shortest path length is: 3`
	`1 4 5 7`

6 7
7 8
1
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