Iterator over Graph

Generated by Doxygen 1.9.1

Chapter 1

Iterator Over Graph

This code starts `iterating` over a graph from the smallest number in each component of it.

1.1 How To Use It

There is no need to select graphs manually. Currently, it automatically loads 7 graphs from Graphs folder. So just type

```
g++ ./main.cpp -o iterator -Wall -Wextra -g -fsanitize=address -lm -lstdc++
```

Then just run it by

```
./iterator
```

1.2 Graph Visualization

There is the graph visualization of the graphs used in this project:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 Graph Struct Reference

`#include <graph.h>`

Collaboration diagram for Graph:

3.2 Iterator Class Reference

Public Member Functions

- **Iterator** (Graph gra)
- void PrintInfo ()

  Print all components and its lowest numbers.
- void ResetStatuses ()
- void Reset (Graph testGrap)

  Process graph.
- void CurrentKey (int id)
- bool IsEnd (std::stack< Node * > st)

  Checks if stack is empty.
- bool IsEnd (std::queue< Node * > qu)

  Checks is queue is empty.
- void DoDFS ()

  Depth First Seacrch.
- void DoBFS ()

  Breadth First Search.

3.2.1 Member Function Documentation

3.2.1.1 CurrentKey()

```
void Iterator::CurrentKey (
            int id )
```

Prints each node's position generated by iterator and also its.

Parameters

id	Id of the node

3.2.1.2 DoBFS()

```
void Iterator::DoBFS ( )
```

Breadth First Search.

DoBFS goes trough graph by using Breadth First Search algorithm. The first step is reseting statuses of all nodes to make sure there are no nodes with other status than visited. Then it starts iterating from the smallest number in each node.

Returns

It prints each node's position generated by iterator and also its ID

3.2.1.3 DoDFS()

```
void Iterator::DoDFS ( )
```

Depth First Seacrch.

DoDFS goes trough graph by using Depth First Search algorithm. The first step is reseting statuses of all nodes to make sure there are no nodes with other status than visited. Then it starts iterating from the smallest number in each node.

3.2.1.4 IsEnd() [1/2]

```
bool Iterator::IsEnd (
            std::queue< Node * > qu )
```

Checks is queue is empty.

processing if the queue, that stores nodes from single component, is empty. If so, it returns TRUE, otherwise its defaulty set as FLASE.

Parameters

qu	Queue, that stores the components from each component

Return values

FALSE	if queue is not empty

Return values

TRUE	if queue is empty

3.2.1.5 IsEnd() [2/2]

```
bool Iterator::IsEnd (
            std::stack< Node * > st )
```

Checks if stack is empty.

processing if the stack, that stores nodes from single component, is empty. If so, it returns TRUE, otherwise its defaulty set as FLASE.

Parameters

st	Stack, that stores the components from each component

Return values

FALSE	if stack is not empty
TRUE	if stack is empty

3.2.1.6 PrintInfo()

```
void Iterator::PrintInfo ( )
```

Print all components and its lowest numbers.

Print each component of the graph and the node thats used as a starting point for following algorithms

3.2.1.7 Reset()

```
void Iterator::Reset (
            Graph testGrap )
```

Process graph.

Reset function goes trough raw graph and it find all components of it. Also it save the smallest number of each component for later use. It does trough graph by using DFS algorithm.

Parameters

testGraph	Raw graph from which the number of components will be calculated

3.2.1.8 ResetStatuses()

```
void Iterator::ResetStatuses ( )
```

Resets statuses for all nodes.

The documentation for this class was generated from the following files:

- iterator.h
- iterator.cpp

3.3 Node Struct Reference

```
#include <graph.h>
```

Public Attributes

- std::vector< Node * > **neighbors**
- int **id**
- int **status** = not_visited

3.3.1 Detailed Description

Each node has vector<Node *> as their neighbors, ID and status

The documentation for this struct was generated from the following file:

- graph.h

Index