Program

August 19, 2021

Contents

Chapter 1

About the program

This program reads a certain description of an undirected graph and determines the number of connected components of it.

1.1 About the description of the graph

The graph shall be read from a file. The file describes the graph by way of specifying the number of nodes, the number of edges and a description of what nodes each edge connects.

1.1.1 Syntax

Referenced by 1 2 3

The input file consists only of a series of arbitrarily space- separated integer constants written in plain ASCII, and consequently, also readable as an UTF-8 plain text file.

The first two integers shall be the number of nodes (n), and the number of edges (m), respectively.

Given this, the set of nodes is assumed to be 1, 2, ... n, and each node will be referred simply by its number.

Then comes "m" pairs of integers "u" and "v" each giving the existence of a bidirectional edge between nodes "u" and "v".

1.2 The command line interface

As a binary, our program shall receive one and only one command line argument, and this will be interpreted as the file name for the input file described in /about/graph-description/syntax .

If the input file doesn't comply with the /about/graph-description/syntax our program will fail silently.

Chapter 2

The program

As this program will be considerably small, it will consist of a single source file in which we will put all of our code.

File: source/app.d The one and only sourcefile. Will contain everything. The structure of this file will be typical.

<source/app.d >+=

```
<Import declarations>
<Global variable definitions>
<Function definitions>
```

Code Fragment: Import declarations In this fragment will be included the import declarations that we see fit.

Code Fragment: Global variable definitions In this fragment we will be creating and initializing global variables as we require them.

Code Fragment: Function definitions In this fragment we will be inserting our function definitions.

As per the rules of the D programming language, the order in which we insert new fragments into the three previous fragments won't affect the final result.

2.1 The main function

The main function will be the entry point to our program. This is the starting point of execution. The structure of this function will be typical of a D program.

<Function definitions >+=

```
void main(string[] args)
{
  <Local variables definitions>
  <Main process>
}
```

Code Fragment: Local variables definitions In this fragment we will be inserting the local variables we require.

Code Fragment: Main process In this fragment we will input how our main function does its processing.

Where 'args' is an array of strings that contains the arguments we received from the user. The first argument is always the name of the binary, the rest are given by the user.

2.1.1 The process

<Main process >+=

```
<Commandline arguments validation>
<Read the input>
<Compute the answer>
<Write the answer>
```

Code Fragment: Commandline arguments validation As it is normal in a program with command-line interface, our first job shall be to validate the arguments that we are given.

Code Fragment: Read the input Given that we have a valid argument giving us the input file name. We will try to open the given file and get it's data (ala Succ). This fragment's job will imply verifying the existence and syntax of the given file.

Code Fragment: Compute the answer With the information about the graph we will now be ready to compute the number of connected components of it.

Code Fragment: Write the answer Given the answer obtained in the previous fragment. We will present it to the user through stdout.

Commandline arguments validation To do this, it is enough to verify that we have been given one and only one user-defined commandline argument. The first given argument in "args" is always the name of the binary, so we need to verify that the args array is of size 2 exactly.

<Commandline arguments validation >+=

```
if (args.length != 2)
{
  import std.stdio: stderr;
  stderr.writeln("Expected an only argument giving an input file");
  return;
}
```

Thus, if we find that we didn't receive the exact required number of arguments, we simply print a message into stderr and exit. %TODO It may be better to have a function that prints a help string.

Before reading the input we must know how we will represent in-memory the given graph. It would seem intuitive to simply map the representation that we know we have in the file (given /about/graph-description/syntax) and proceed from there, but for our purposes, it will prove to be more convenient to represent it other way.

Graph representation

There are 3 standard ways of representing a graph, suppose we have n nodes and m edges:

Adjacency matrix representation In this representation we have an $n \times n$ matrix such that $a_{ij} = 1$ if i and j are connected directly by an edge and $a_{ij} = 0$ otherwise.