Documentation for CPSC 2150 Assignment 10

Gladys Monagan
Department of Computing Science and Information Systems
Langara College

March 31, 2021

Abstract

In order to give the CPSC 2150 a chance to concentrate on implementing the data structure *list of neighbours* and on using that data structue to solve problems by implementing algorithms, I wrote a "menu style" test program and I am supplying some test files.

1 Running doGraph

1.1 running the program interactively

The program doGraph can be run interactively (as it reads from std::cin) by typing in the commands. Run the program on Linux

```
./doGraph
```

on Windows

```
doGraph.exe
```

1.2 batch mode

The other option is to pass a predefined line argument `-batch` that outputs every character read and outpus the results. Presumably the input will come from a file e.g. *input.txt*. Run the program on Linux

```
./doGraph -batch < input.txt
```

on Windows (use CMD not the power shell)

```
doGraph.exe -batch < input.txt
```

2 Commands of the program doGraph

When running the program in the non-batch mode, the user can see the menu of commands

```
 --------------choose a command  -------------------------------
(i) - input the file name that contains the graph
(o) - output the graph to std::cout
(n) - number of vertices in the graph
(c) - determines whether the graph is connected or not
(f) - find a cycle
(l) - list the connected components (one set of vertices per line)
(t) - twin the graph by making a copy
(d) - done with the graph
(m) - mention or comment, the subsequent line is ignored
(q) - quit the program altogether
(p) - calculate the path length between two vertices
-----------------------------------------------------------------
```

Only the first letter of the command is used but the user could type the complete word. For instance, instead of `i`, the user can type `input`

2.1 input

`input` is for entering the data from a file to become a graph.
The user is queried for a file name e.g *g1.txt* as provided with the assignment.
`inputGraph` opens the input stream, making sure that the file is readable and reports errors as needed.

2.1.1 to do as part of the assignment

Implement in Graph.cpp the overloaded assignment operator >>
Read into the instance of the class Graph the information from the open stream, building a list of neighbours using a dynamic array.
Do not use the STL.
Note that it is required that the previous graph be deallocated first.

2.2 output

Uses std::cout to output the graph. It is up to you how you want to display the graph.

2.2.1 to do as part of the assignment

Implement in Graph.cpp the overloaded assignment operator <<

2.3 number

gives the number of vertices in the current graph

2.3.1 to do as part of the assignment

Implement in Graph.cpp the accessor method (function) `numberOfVertices`

2.4 connected

Outputs whether the graph is connected or not.

2.4.1 to do as part of the assignment

Implement in Graph.cpp the function `isConnected`
You must use a **breadth first search** approach.

2.5 find a cycle

Outputs whether the graph has at least one cycle or not.

2.5.1 to do as part of the assignment

Implement in Graph.cpp the function `hasCycle`

2.6 list the connected component

A graph can have several connected components. This command outputs *all* the connected components by listing the vertices of each connected component. Each connected component is in a separte line.

For instance, Figure 1 has 2 connected components. One connected component is the set of vertices $\{0, 3\}$ and the other component is the set of vertices $\{1, 6, 4, 5, 2\}$.

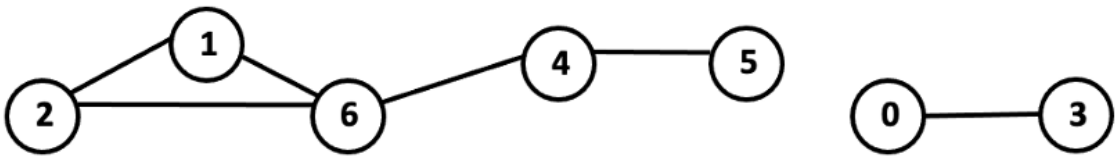

Figure 1: file g0.txt

2.6.1 to do as part of the assignment

Implement in Graph.cpp the function `listComponents` that outputs using the passed output stream.
List each set of vertices one per line.
A single vertex, without any edges adjacent to it, is a connected component and needs to be listed in a single line.

Example, not necessarily in this order, the output for Figure 1 could be

```
0 3
1 6 4 5 2
```

2.7 twin the graph

Twinning the graph simply means making a copy of a graph. The copy is manipulated with the same commands. Exit the twin mode, with `done`.

2.7.1 nothing to do as part of the assignment

`twin` tests the copy constructor and the destructor.

2.8 done

Indicates that the user is done with the graph.
Presumably the user wants to read another graph.
`done` is also used to stop the twinning mode.

2.8.1 nothing to do as part of the assignment

2.9 quit

Exit `doGraph`

2.9.1 nothing to do as part of the assignment

2.10 mention

Mention or document or make a remark in the next line.
This is useful in the input file to indicate something about the graphs being read.

2.10.1 nothing to do as part of the assignment

2.11 BONUS: path length calculation

Calculate the length of the path between two vertices.
The user is queried to enter the source vertex and the destination vertex.

If the source and the destination vertices are the same, the path length is 0.
If there is no path from the source vertex to the destination vertex, -1 is printed.
If the vertex entered is not a vertex of the graph, -1 is printed.

2.11.1 to do as part of the assignment

Implement in Graph.cpp the function `pathLength`

3 Format of the Input Files provided

The first integer is the number of vertices in the graph say n. The subsequent pairs of numbers correspond to unordered edges. We don't know how many edges there are so we continue reading until the input stream enters the fail state (be it because a non numeric character is encountered or, it is the end of the file).

There are no self-loops nor are there parallel edges in our representation of a graph. The vertices are always 0 1 2 3 ... $n-1$.

3.0.1 to do as part of the assignment

When inserting an edge into the graph, check that both vertices of the edge are in the range of possible values. If the edge is already there, do not enter it again.

3.1 Example g1.txt

```
7
0 1
1 2
2 3
2 4
4 3
```

- there are 7 vertices in the graph
- the vertices are numbered 0 1 2 3 4 5 6
- there are 5 unordered edges in this graph: edge 0 1, edge 1 2, edge 2 3, edge 2 4, edge 4, 3
- there are two single vertices not connected to anything, namely the vertex 5 and the vertex 6