Graph Representation and Cycle Detection

Introduction

Problems that can be represented with a graph and an algorithm show up frequently in Computer Science. In this project, we will try two different ways to represent graphs:

- Matrix representation
- Adjacency list representation

Each representation have strong and weak spots, when considering space and operations, also quite dependent on the characteristics of the graph (sparse or dense). Knowledge on this come in handy in operating systems, when we study access matrices.

The problem we will use the graphs for are detecting cycles, as this can be usefull, for example in determining if an operating system is in a deadlocked state.

Representing the Graph

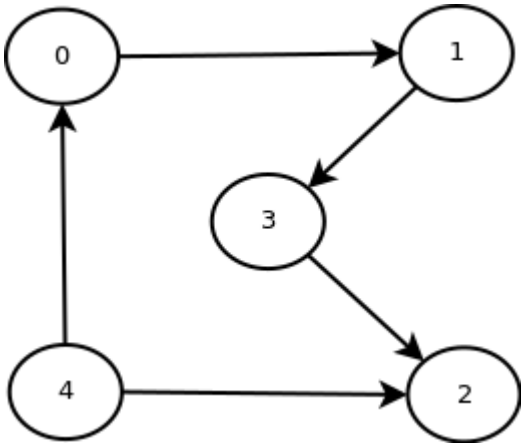
Matrix Representation

The input graphs that you will be working on is in matrix representation. The first line has just an integer, that tells the number of vertices are present in the graph. The next lines are the matrix, where a one tels us there is a directed edge from the node on the line we are on, to the edge in the column we are in.

A small example graph:

```
5
01000
00010
00000
00100
10100
```

Which will be this graph:



Adjacency List Representation

You must store the graph in your program using the adjacency list representation. In this representation, we only store the edges that are there, thus all the zeros in the matrix representation are not stored.

In each vertex, you could just store a reference to the vertices that are pointed to, but as we also need a fast way to determine who points to me, we store both a reference to vertices we are pointing to, and let each of those vertices have a reference to the vertices that points to them.

In the skeleton project, the header files for the graph and the vertex provides more input to how you can store the graphs.

Cycle Detection

For cycle detection, you should use **Kahn's algorithm**, which generates a topological sort in time **O(|V|+|E|)**. A topological sort is only possible if the graph does not have any cycles.

```c
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

If the graph is a DAG, a solution will be contained in the list L (the solution is not necessarily unique). Otherwise, the graph must have at least one cycle and therefore a topological sorting is impossible.

You are allowed to destroy the graph in the proces of finding the DAG.

Requirements

Makefile

A Makefile is provided in the skeleton project. This should be used to output the binary, named `program`

Input and Output

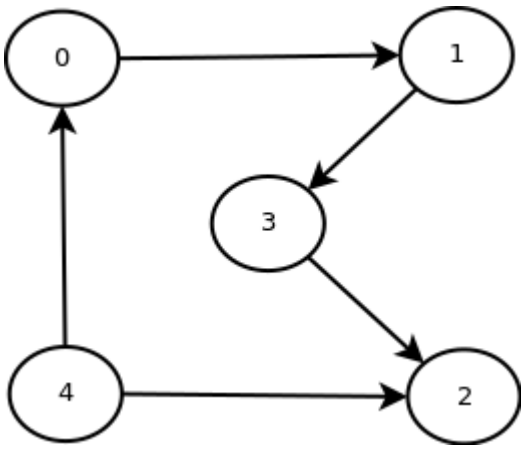
Your program should take the inputfile as its single parameter. You should be able to run the compiled program as

```
./program ../graphs/graphmatrix-1.txt
```

```
after compiling the program in the `src` directory
```

Example

The graph below



Should have the vertices listed as a DAG, so the output should be

```
4, 0, 1, 3, 2
```

If a graph contains a cycle, the output should be:

```
CYCLE DETECTED!
```

Both of the outputs should be followed by a single newline character '**\n**'

Checking your solution

To verify your solution, you can use use the checker supplied with the skeleton project.

In the root of the project, use

```
./gradlew checkProgram
```

Each of the 9 tests will indicate if they have passed or not. The last of the input graphs: *graphmatrix-10.txt* does not have cycles, but the DAG is not unique, so it is not included in the automated tests.

Getting Started

Suggested steps to get started:

- Download the skeleton project, that includes header files, Makefile, graphs to test on etc.
- Create the linked list that you need to store the vertices and in the algorithm.
- Read in the graph (specified on the command line) into a datastructure
- Printout the graph (for debugging, remove this once you proceed).
- Implement Kahn's algorithm
  - You can remove edges from the graph in the proces.
- Output the result

Report

The primary objective of this project is to program a lot of c-code.

You do not have to write a report, but should include a README file in your project, shortly describing the status of your implementation. It should also describve if there is corner-cases you do not handle, did you not complete everything, etc.

You should instead of the report focus on writing well formatted, easy understandable and maintainable code. To do so, make sure formatting is consistent, you use descriptive variable names, the code does not have compile warnings, you refactor large methods into smaller

ones etc.

What to turn in?

You must make sure the program compiles and runs on an IMADA machine in the terminal room, and you have a file called README in your source folder as described above.

In your project source folder, run **make clean**. Place yourself above the Linux directory with these files and pack it with

```
tar zcvf <your name>.tgz <directory name>
```

and upload the resulting `.tgz` file

Make sure you hand in the project, even if you did not complete it. I may approve projects if I find a large enough efford to learn c-programming, even if the project does not pass all tests

Extra Challenges

The following are suggestions for extra challenges, that are not required for succesfull completion of the project

- Remove memory leaks/clean up your memory. This requires to free alle malloc'ed memory, typically in the backwards order.

F.A.Q.

No questions yet :)