DM548/DM588
Computer Architecture
(& System Programming)

Fall 2023

Assignment 2: Cycle Detection in C

This assignment contains only a programming element.

Many problems in Computer Science can be solved by modelling them using graphs and then using suitable algorithms. In this assignment you are must implement a C program for detecting cycles in graphs.

1 Representing Graphs

In this assignment we will use two different ways to represent directed graphs:

- Adjacency matrix representation
- Adjacency list representation

Each representation has strong and weak properties. Which one to use dependents both on the characteristics of the graph being represented (sparse or dense), and on which operations will be used.

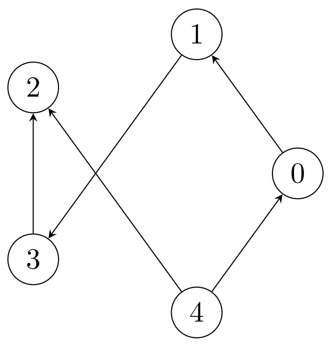
Adjacency Matrix Representation

The input graphs that you will be working on are stored in a textual adjacency matrix representation. The first line in a graph file has just an integer n, which tells the number of vertices in the graph. The next n lines encode the matrix, where the character in line i column j tells us whether or not there is a directed edge from vertex i to vertex j. An ASCII character `'1'` means the edge is there and a `'0'` means that no such edge exists.

Example The input

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

encodes the graph



A number of test graphs are provided.

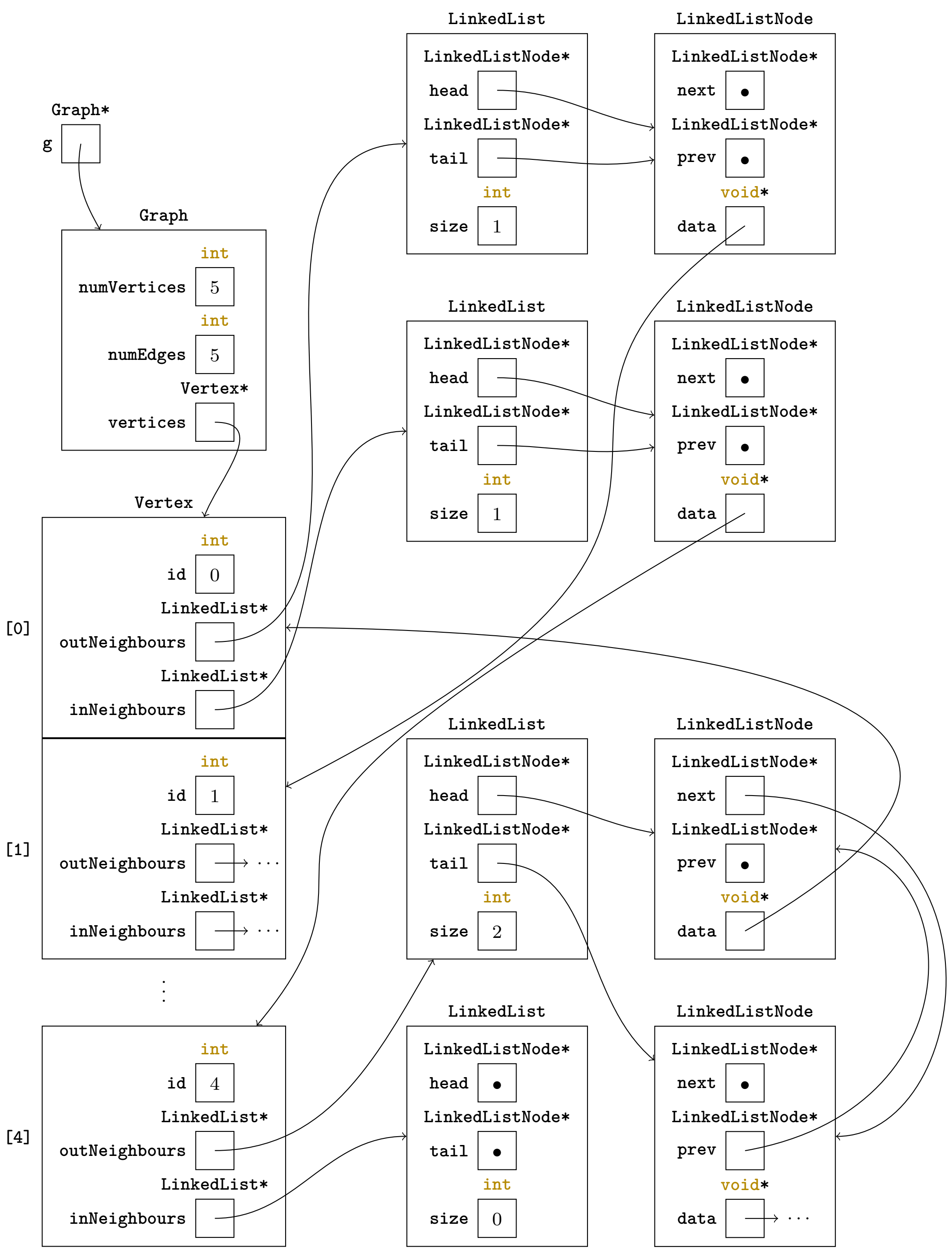
Adjacency List Representation

You must store the graph in your program using an 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 explicitly.

In each vertex, we could just store a list of references to the vertices that are pointed to (i.e., a list of out-edges). However, we also need a fast way to determine which vertices points into a given vertex as well, so we must store another list of vertex references, to those vertices that points to us (i.e. a list of in-edges).

The given set of files contains header files with definitions of a graph and vertex, which provides more hints on how you can store the graphs.

Example Using the example graph above, and the data structure definitions in the given code, the in-memory data structure looks similar to the following.



Graph*
g
Graph
int
numVertices
5
int
numEdges
5
Vertex*
vertices
Vertex
[0]
int
id
0
LinkedList*
outNeighbours
LinkedList*
inNeighbours
[1]
int
id
1
LinkedList*
outNeighbours
···
LinkedList*
inNeighbours
···
[4]
int
id
4
LinkedList*
outNeighbours
LinkedList*
inNeighbours
LinkedList
LinkedListNode*
head
LinkedListNode*
tail
int
size
1
LinkedListNode
LinkedListNode*
next
LinkedListNode*
prev
void*
data
LinkedList
LinkedListNode*
head
LinkedListNode*
tail
int
size
1
LinkedListNode
LinkedListNode*
next
LinkedListNode*
prev
void*
data
LinkedList
LinkedListNode*
head
LinkedListNode*
tail
int
size
2
LinkedListNode
LinkedListNode*
next
LinkedListNode*
prev
void*
data
LinkedList
LinkedListNode*
head
LinkedListNode*
tail
int
size
0
LinkedListNode
LinkedListNode*
next
LinkedListNode*
prev
void*
data
···

2 Cycle Detection

For cycle detection, you must 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. The following pseudo code describes the algorithm:

```
Data: An input graph $G$.
Result: The vertices of $G$ in topological sorted order, or an error.
$L \leftarrow$ an empty list of vertices
$S \leftarrow$ a set of all vertices of $G$ with no incoming edges
while $S$ is non-empty do
    $u \leftarrow$ a node removed from $S$
    Append $u$ to the tail of $L$
    foreach vertex $v$ in $G$ with an edge $e$ from $u$ to $v$ do
        if $v$ has no other incoming edges than $e$ then
            Insert $v$ in $S$
        Remove edge $e$ from $G$
if $G$ has any edges left then
    return error // the input graph had at least one cycle
else
    return $L$ // a topological sorted order of the vertices
```

If the graph is a DAG (directed acyclic graph), a topological ordering of the vertices 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.

Note that the algorithm destroys the graph in the process, and your implementation may do the same.

3 Input/Output Specification

You must provide a makefile (see submission format) which produces a program from you sources called `detectCycles`. Your program must be callable with a single argument, which is the filename for an input file, e.g.:

```
> ./detectCycles graphmatrix-1.txt
```

If the graph is a DAG the output must be the list of vertices in a topological order, e.g., for the graph above:

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

If the graph is not a DAG, i.e., it contains a cycle, the output must be

```
CYCLE DETECTED!
```

In both cases the output must appear as a line, i.e., it must end with a newline character `'\n'`. Both cases are considered as successful, i.e., the exit code must be 0, and thus no other output than the specified must appear on stdout. If the program can not carry out its task (e.g., bad/missing input or out-of-memory) the exit code must be non-zero. You may assume that input files are always in the correct format.

Practicalities

Given Code

Your designated assignment 2 repository should already contain a number of test instances, a `main.c` file, and a bunch of header files that defines the needed data structures and declares a number of functions to be implemented. You are welcome to alter the given files in any way you see fit. Though the graph data structure you implement must be an adjacency list in some form.

Working in Groups

You may not work in groups.

Submission Format

Submission is done simply by pushing to your designated assignment 2 repository. There will be automated checks and feedback of your repository contents. You can see the status at `https://dalila.imada.sdu.dk` after logging in with your SDU credentials.[1] While you should aim to make all checks succeed, note that they do not cover all aspects of the assignment, and their status therefore does not alone determine how your submission is evaluated. The automated tests are still of experimental nature, so additional checks may be added, and please do contact the lecturer if you find errors or misleading results.

The repository must have the following content:

- `src/`
  A folder where all your source code files must directly reside in, i.e., both your header (`.h`) and implementation (`.c`) files, and no subdirectories.

- `src/Makefile`
  A makefile which can be used to create your cycle detection program. Specifically:

[1]See also the general note on automated testing in the course, available on the course website.

- The default target must be made such that when executing “`make`” inside the `src/` folder, it must produce an executable “`detectCycles`” in that same folder.
- A target “`clean`” must exist that deletes all files that can be generated from the makefile. Typically that will be all object files and all executables that the makefile can generate.

Submission and Grading

Submission is simply done by pushing to your designated assignment 2 repository. At the deadline the current state on the Git server will be assumed to be your submission. The final deadline is

10:00 (CET) Monday, 18 December, 2023

Late submissions are not accepted, and submission through other means than your designated repository will not be accepted either.

This assignment will not be evaluated by it self, but will be taken into account in an overall evaluation of the two assignments and the final written test.