DEPARTMENT OF COMPUTING AND INFORMATION SYSTEMS

COMP20007 Design of Algorithms, Semester 1, 2016

Assignment 1: Topological Sort

Due: 5pm Friday 8 April

Objective

Suppose that you are undertaking a complex task that is made up of many sub-tasks, each with dependencies. Examples of this can easily be found, e.g. baking a cake, constructing a building, doing a university degree, ... Note that some of the sub-tasks must be completed before others can begin. This situation can be represented using a directed graph, where tasks are represented as nodes, and dependencies are represented as directed edges between the nodes. Thus, an edge (t_i, t_j) expresses the situation that task t_i must be completed before task t_j can begin.

The topological sort, or *toposort*, of a connected directed acyclic graph is a list of the n nodes of the graph $t_0, t_1, \ldots, t_{n-1}$ such that for every directed edge (t_i, t_j) in the graph, t_i precedes t_j in the list.

By computing the topological sort of a graph of dependencies, we discover a sequence which satisfies the dependencies.

In this project you will implement two algorithms for topological sorting of directed graphs, and study their performance.

Files: Please download the skeleton code from the LMS. **Do not alter filenames or structure**. The files are as follows.

`Makefile`	Download from LMS and edit to include your student id.
`main.c`	Download from LMS and do not change.
`graph.h`	”
`list.h, list.c`	”
`graphio.h`	”
`toposort.h`	”
`graphio.c`	Replace with your own implementation.
`graph.c`	” (based on your lab work)
`toposort.c`	”

Note that you should be able to compile the supplied code, although you will see warnings about unused parameters.

Once your project is completed, you will be able to invoke it as follows:

```
ass1 -p mygraph.dot mygraph.txt                 # prints a directed graph
ass1 -m 1 mygraph.txt > mygraph_toposort1.txt    # performs toposort (method 1)
ass1 -m 2 mygraph.txt > mygraph_toposort2.txt    # performs toposort (method 2)
ass1 -v mygraph.txt < mygraph_toposort2.txt      # verifies a toposort
```

Project

The project consists of five parts, one for graph input, three for algorithms, and one for your report. Each part is worth 2 marks, for a total of 10 marks.

Part 1: Graph Input and Output

Implement function `load_graph()` in `graphio.c` to load a graph from a plain text file. The file consists of a line specifying the number n of vertices in the graph, followed by a sequence of n lines being the unique labels of these vertices (less than 256 characters long), followed by a sequence of ordered pairs to indicate directed edges between the vertices. Vertex labels may include punctuation and spaces. You may wish to use the function `fgets()` in the standard C library for reading vertex labels. Vertices are to be identified using the integers $0 \dots n-1$. Here is an example of the file format:

```
3
COMP10001
COMP10002
COMP20007
0 1
0 2
1 2
```

You can assume that your program will only be given input that conforms with this format specification, and will correspond to a labelled directed graph.

Implement the function `print_graph()` in `graphio.c` to ensure that you have correctly read in the graph. `print_graph` outputs the graph to the DOT file specified in the `-p` command line argument. The DOT file corresponding to the above input is shown below:

```
digraph {
    COMP10001 -> { COMP10002 COMP20007 }
    COMP10002 -> { COMP20007 }
}
```

The program graphviz can be used to open files in the DOT format to visualise them. If graphviz is installed and available in the `PATH` there is a target in the `Makefile` that converts a dot file to a PNG.

Part 2: DFS Algorithm for Toposort

Implement the DFS algorithm for toposort from Wikipedia.[1] (Use auxiliary arrays of Boolean values to mark temporarily and permanently visited vertices.) Do your work in `toposort.c`, in a function named `dfs_sort()`. `dfs_sort()` takes a `Graph` as input and returns a list of vertices. Graph and Vertex are defined in `graph.h`. Lists are defined in `list.h`. You will be able to run this sort algorithm using the `-m 1` command line option.

If it was not possible to provide a topological sort of the graph, for any reason, you should report the error to the `stderr` stream and your program should terminate execution and return `EXIT_FAILURE` to indicate failure.

[1] `https://en.wikipedia.org/wiki/Topological_sorting`

Part 3: Kahn's Algorithm for Toposort

Implement Kahn's Algorithm for toposort, also from Wikipedia. As before, do your work in `toposort.c` in a function `kahn_sort()`. `kahn_sort()` takes a `Graph` as input returns a list of vertices. You will be able to run this sort algorithm using the `-m 2` command line option.

As before, if it was not possible to provide a topological sort of the graph, for any reason, you should report the error to the `stderr` stream and your program should terminate execution and return `EXIT_FAILURE` to indicate failure.

Part 4: Verification

Design and implement an algorithm to verify that a given sequence of vertices is a correct topological sort of a graph. Extend `toposort.c` using a function named `verify()`, which takes a list of vertices as input and returns a Boolean value. You will be able to run this verification function using the `-v` command line option.

Part 5: Analysis

Write a two-page report (11 point font) on the efficiency of both algorithms. A good report should include: a graph of running times on random inputs of increasing size; a discussion of the curves in the graphs relative to the big-O running times of the algorithms; and some times (perhaps in a table, perhaps a graph) of the two programs on pathological inputs, again with a discussion relating them to the big-O bounds. Submit your report in PDF format, as a file named `report.pdf`.

Submission

Submission is via LMS under COMP20007, "Assignment 1". Submissions will close automatically at the deadline. NB. Machine and network load/problems right before the deadline is not a sufficient excuse for a poor or missing submission. As per the Subject Information Sheet from Lecture 1: "The late penalty is 20% of the available marks for that project for each day (or part thereof) overdue. Submissions more than three days late will not be accepted."

Submit a single archive file (e.g. `.tar.gz` or `.zip`) via the LMS containing the four files: `toposort.c`, `graphio.c`, `graph.c`, `report.pdf`. It should unpack into a folder, where the folder is named using your student id. To make this easy, we've added a submission target to the Makefile. You need to locate the line `STUDENTID = <STUDENT-ID>` and modify it with your student id. Then you can type `make submission` to create the required archive file.

Submissions not adhering to these requirements will be subject to a 2 point penalty.

Testing

We provide some basic tools for you to test your implementation. Three small graphs, with their corresponding dot files, are provided with the skeleton code. You may use these to verify basic functionality of your program.

- The dependency graph for the program;
- A course plan for B-Sci (Computing and Software Systems); and
- Clement's course plan at Melbourne University.

'Also a simple program to generate random directed acyclic graphs is provided in the test folder and may be compiled with the command `make daggen`.

It takes up to two arguments. The first argument is the number of vertices in the generated graph (default 10). The second is the probability there is an edge from one vertex to a subseqent vertex (default 35%).

The daggen program may create a graph of 100 vertices with 50% chance there is an edge between vertices by invoking `daggen 100 50`

Marking: The four coding parts of the assignment will be marked as follows: you will lose a mark if your submission does not identify you in the opening comment of each submitted file; you will lose a mark if your solution is incorrect in some way (breaks on certain inputs, has memory leaks, requires fixing to work at all); and you will lose a mark if your solution is difficult to interpret (minimal or unhelpful comments, obscure variable names). The report will get full marks if it is a succinct, factual report on running times including a discussion comparing times to big-O bounds. You will lose marks for missing timing results, missing graph, missing big-O discussion. *NB, the assessment is on the algorithmic content of your work and your effective communication; this is not primarily a programming assignment.*

Academic honesty: All work is to be done on an individual basis. Any code sourced from third parties must be attributed. Please see the Subject Information Sheet for more information. Where academic misconduct is detected, students will be referred to the School of Engineering for handling under the University Discipline procedures.