COMP26120 Lab Exercise 11
Graph Representation and Traversal

Duration: 1 lab session

For this assignment, you can only get full credit if you do not use code from outside sources.

Aims

To understand and be able to implement data structures to represent graphs, and traversal algorithms to search the part of the graph connected to a given node.

Learning Outcomes

On successful completion of this exercise, a student will:

1. Have developed a data structure to represent directed graphs using adjacency lists.
2. Will have implemented depth-first, breadth-first (optionally both) traversal algorithms.
3. Will be able to explain the advantages of adjacency lists over adjacency matrix representation.

Summary

You will be given some data files containing descriptions of directed graphs, and some code to read this into your data structure. You will produce a data structure which can represent a directed graph. You will then produce the following algorithms:

1. an algorithm which finds the node with the largest out-degree.
2. an algorithm which finds the node with the largest in-degree.
3. an algorithm which takes a node as input, and finds the node with the largest out-degree which can be reached from that input node.

You will also be asked to discuss the computational advantages of the adjacency list representation over the adjacency matrix representation for these tasks.

Deadlines: The unextended deadline is the end of your scheduled lab. If you need it, if you attend the lab you will get an automatic extension to your next COMP26120 lab session (you must use 'submit' to prove you finished in time and get it marked at the start of your next scheduled lab). You can also get an extension for good reason e.g. medical problems.

Description

This lab is intended to give you a gentle introduction to algorithms on graphs. If you are taking COMP26120, the data structure and the algorithm you develop in this lab will be used and extended in the next lab, so it is important that you do this lab on time in order for you to be able to carry on with the later labs.

We will be working with "directed graphs". This means that the edges have a direction. A edge points from a source node to a target node. An adjacency list representation associates with each node a list of the nodes which are targets for the node's edges.

To get you started, we will give you the representation of a node. This can be found in the file `graph.h`.

A node is a structure which contains fields for the following: its name (a string), an adjacency list of indices of the nodes it has links pointing to (a linked list of int), its out-degree, which is the length of the adjacency list (an int) and a field for its pagerank score. Not all of these fields are needed for this lab; some will be used in the next lab.

A graph is a collection of nodes. You will set up the data structure for this. For this lab, the size of graph (the number of nodes it contains) will be given as the first piece of information about the graph. Thus, you know when the graph is initialized how much memory will be required. You will need to produce code to initialize a graph, and add nodes and add edges. Put this in the file `graph_functions.c`.

Produce code to find the following three nodes: the node with the largest out-degree, the node with the largest in-degree, and the node with the smallest (non-zero) in-degree. Put this code in `part1.c`. This part should be very easy, as you just need loop through the graph an appropriate number of times. Think about, and be prepared to answer how these tasks would be different had we used adjacency matrix representation of the graph. Which would be easy or run faster. Which would be harder or run slower.

Produce functions to traverse the graph. This should take as input a node index, and search all the nodes reachable from that node. Use this code to answer the following questions:

1. Is the node with the largest out-degree reachable from the node with the smallest (non-zero) out-degree?
2. If not, what is the largest out-degree of a node which can be reached from the node with the smallest (non-zero) out-degree of all the nodes in the graph?
3. Can the entire graph be reached from the node with the smallest (non-zero) out-degree?

(Each question above is written as if it has a unique answer e.g. "**the** node with ...". If a graph contains more than one such node, then you can arbitrarily pick any one of them - e.g. the first such node that you find.)

For full credit, produce a depth-first search and a breadth-first search traversal. For depth-first search, you can use the recursive algorithm described in the lecture. For breadth-first search, you need to maintain a search queue and a list of explored nodes. To carry out a step of search, you get and remove the first node from the search queue and process it. You then get its adjacency list, and check each node to see if it is in the explored list. If it is not, add it to the back of the queue, and to the explored list. When the search list is empty, the algorithm will have explored all the nodes reachable from the starting node.

In the above description of breadth-first search, the search list is a queue; you remove from the front and add to the back. I.e. it is first-in first-out (FIFO). If you replace the search list by stack in which you remove from the front and add to the front (FILO), the traversal becomes depth-first search. Thus, you can implement depth-first search and breadth-first search by sharing most of the code, and use conditional compilation to determine which parts of the code get compiled. I.e. surround the code which adds to the search list at the back by:

```
#ifdef BFS
...
#endif
```

and the bit which adds to the front by:

```
#ifdef DFS
...
#endif
```

Then put at the start of your file containing the search functions:

```
#define BFS
#undef DFS
```

to get breadth-first search, and

```
#define DFS
#undef BFS
```

to get depth-first search. Put the search code in the file `graph_search.c`. Put the main for this part in `part2.c`. (We separate them into two files because you will reuse the graph and search functions in later labs.)

Is it possible to say which is better for this task, breadth-first or depth first search?

In the directory `/opt/info/courses/COMP26120/problems/ex11/data` you will find files with the suffix .gx. These contain descriptions of graphs, using a simple format. Here is an example:

```
MAX 10
NODE 1 one
EDGE 1 2
EDGE 1 3
EDGE 1 1
NODE 3 two
EDGE 3 5
EDGE 3 1
NODE 5 three
EDGE 5 7
EDGE 5 1
NODE 7 four
EDGE 7 7
NODE 9 five
NODE 2 six
```

The first line "MAX 10" tells that the node will have indices 1-10. The first line of these graphical files will always have such a line. Lines of the form "NODE 1 name" defines node number 1 with name "name". Lines of the form "EDGE 3 5" define an edge from node 3 to node 5. So, in the above example, Node 1 has as its adjacency list (1,2,3) since there are edges from 1 to 1, 1 to 2 and 1 to 3. The order of these lines is unimportant except the MAX line must come first. We have given you code which reads in such files, which can be found in `graph_functions.c`.

Summary of Tasks

1. Modify `graph.h` so it contains the definition of data type GRAPH.
2. Modify `graph_functions.c` so the functions `initialize_graph`, `insert_graph_node` and `insert_graph_link` are implemented. If you change the prototypes, be sure to change the calls in `read_graph`.
3. Produce `part1.c` to run through the graph looking for the nodes with the largest and smallest (non-zero) in-degrees and out-degrees. Run `make part1` to make this.
4. Think about the question of how the code and run-times would be different if you used adjacency matrix representation.
5. Produce `graph_search.c` with the graph traversal functions (depth-first search, breadth-first search or both) and `part2.c` containing the main which calls this to find the node with the largest outdegree which can be reached from the node with the smallest (non-zero) out-degree, and count the number of nodes reachable from the node with the smallest (non-zero) out-degree. Run `make part2` to make this. Note that this does not recompile `graph_functions.c` if this was not changed.
6. Is performance any different for depth-first and breadth-first search?

Marking Process

You should have in your directory COMP26120/ex11 the following files: graph.h, graph_functions.c, graph_search.c, part1.c and part2.c. The latter two contain mains; all the functions will be in the first three. The Makefile allows you to compile these separately.

The marks are awarded as follows:

```
Graph representation:
 1 mark: completing the representation of the graph
 2 marks: code to initialize the graph, insert nodes and edges.
Graph search:
 1 mark: simple code to find nodes with extremal degrees.
 2 marks: depth-first search traversal
```

```
  2 marks: breadth-first search traversal
 Understanding:
  2 marks: answering questions about efficiency of
           adjacency list versus adjacency matrix representation, and
           breadth-first versus depth-first search.
 Total 10
```