MTE 140 - Spring 2009. Project 4.

Digraphs. Due Date: Friday July 24th, 11:59pm.

Hanan Ayad

1 Project Description

In this project, you will implement a directed graph (digraph) that is unweighted (no edge weights). You are assigned the task of defining an appropriate representation for the digraph, other data structures and their associated algorithms, as needed. Furthermore, it is required that your implementation satisfies certain efficiency requirements.

As part of this project, you will implement a topological sorting algorithm for finding a topological ordering of the digraph vertices, provided that the digraph is acyclic (DAG). The running time of the algorithm, as seen in class, is $O(n + m)$, where n is the number of vertices and m is the number of edges. It is required that your implementation be particularly suited for sparse digraphs, where m is expected to be much less than the possible number of edges, $m << n(n-1)$. So, the representation should speed up the running time of the topological sorting algorithm, enabling it to be less than $O(n^2)$, when the graph is sparse. Furthermore, the space requirements for the the graph representation should be $O(n + m)$, which would also be less than $O(n^2)$, provided that the graph is sparse.

In your comments for the topological sorting algorithm, briefly outline how the required running time is achieved. Also, in your comments for the definition of the digraph representation, outline how the space requirements are achieved. In your comments for other digraph operations, point out the tradeoffs that may be associated with the chosen representation, if any. That is, would certain operations be simpler or more efficient, if an alternative digraph representation is used?

The following source files are given.

1. The interface (header) file `digraph.h`, which has the function declarations for the digraph.
2. An incomplete implementation file `digraph.c`, in which you are required to define a suitable digraph representation, include other data structures, as needed, and implement the body of each of the declared functions.

Note: For this project, you may add other source files that you need for your project. It is required that you apply modular design and data abstraction concepts. You may re-use your own code from previous projects. Include the authorship header in all your submitted source files. You can also add other internal functions for the digraph, as needed.

2 Specifications of the Digraph Operations

The functions you will be implementing for the digraph are described as follows:

1. `DiGraph digraph_create( unsigned int n )`
   This function takes as an argument the number of vertices n and creates a digraph with n vertices numbered 0 to $n-1$. If n is zero, a default of 50 vertices is used. The function makes appropriate initializations, as needed, and returns the digraph pointer (NULL if memory allocation fails).

2. `void digraph_destroy( DiGraph DG )`
   This function takes as arguments a DiGraph DG and frees all allocated memory for it.

3. `int edge_insert( DiGraph DG, vertex u, vertex v )`
   This function takes as arguments a DiGraph DG and two vertices u and v and inserts a directed edge (u, v) from vertex u to vertex v. It returns 1 when insertion is made, 0 if u or v does not exist in the digraph, and -1 if DG is NULL.

4. `int edge_remove( DiGraph DG, vertex u, vertex v )`
   This function takes as arguments a DiGraph DG and two vertices u and v and removes the directed edge (u, v) from vertex u to vertex v is it exists. It returns 1 when removal is made, 0 if u or v or the edge (u, v) does not exist in the digraph, and -1 if DG is NULL.

5. `int vertex_outdegree( DiGraph DG, vertex v )`
   This function takes as arguments a DiGraph DG and a vertex v. It returns the out-degree of vertex v if v exists in DG. It returns -1 if v doesn't exist in DG or if DG is NULL.

6. `int vertex_indegree( DiGraph DG, vertex v )`
   This function takes as arguments a DiGraph DG and a vertex v. It returns the in-degree of vertex v if v exists in DG. It returns -1 if v doesn't exist in DG or if DG is NULL.

7. `int adjacent( DiGraph DG, vertex u, vertex v )`
   This function takes as arguments a DiGraph DG and two vertices u and v. It returns 1 if there exists a directed edge (u, v) from u to v in DG and 0 if such edge does not exist. It returns -1 if any of the input vertices does not exist or if DG is NULL.

8. `int edge_count( DiGraph DG )`
   This function takes as an argument a DiGraph DG and returns the number of edges in DG. It returns -1 if DG is NULL.

9. `void digraph_print ( DiGraph DG )`
   This function takes as an argument a DiGraph DG and prints for each vertex v from 0 to $n - 1$ the successors of v.

10. `int topological_sort( DiGraph DG )`
    This function takes as an argument a DiGraph DG and prints a topological ordering of the vertices if DG is a DAG (Directed Acyclic Graph), returning 1 in this case. The topological ordering printed by this function is based on a **breadth-first** traversal. The function returns 0 if DG is cyclic and -1 if DG is NULL.

3 Documentation and Programming Style

It is required that your name and ID appear at the top of all your submitted text files. Your code should be clear and understandable; you are required to use proper spacing and indentation to enhance the clarity of your code, use meaningful variable names, and useful comments. You should include comments for each function indicating its specifications. This includes a description of what it does, its parameters, return values, and any assumptions that it makes.

4 Submission Requirements

For the submission of this project, you are required to follow the steps outlined below:

1. Create a directory with the name `uwuserid_NNNNNNNNp4`, where `uwuserid` is your UW user ID, `NNNNNNNN` is your UW student ID number, and p4 denotes project 4.

2. Save all the sources files under this directory (do not create subdirectories). The files include: `digraph.h` and `digraph.c`, and any other source files you may need.

3. Implement the project and generate all the object (.o) files. You are not required to submit a `main` function for this project (but you will need to write one for yourself to test your functions).

4. Zip the directory into one .zip file with the same name. That is, `uwuserid_NNNNNNNNp4.zip`. The unzipping should result in creating a subdirectory under test with the name `uwuserid_NNNNNNNNp4`, with all the project files being extracted under this subdirectory.

5. Submit the file `uwuserid_NNNNNNNNp4.zip` into the dropbox designated for project 4 on UW-ACE (under MTE 140).