CSE 101 Winter 2018
Programming Assignment (PA) One

Due: Friday January 26, 11:59 PM

Link to starter code: https://github.com/UCSD-CSE101/W18-PA1

These four questions implement some of the algorithms discussed in lecture so that you are comfortable with their usage and with applying them to future problems. Please remember, do not print in the final versions of the files you will be turning in.

Question 1: Depth-first Search [10 Points] :: DFS.cpp/hpp

Input: a directed graph and a particular vertex ID
Output: set of all vertices reachable from the given vertex ID

A vertex v is reachable from u if there is a directed path from u->v.
The set of vertices reachable from v includes v itself.

Starting things off, you will implement a depth-first search algorithm using the provided headers for Graph and Vertex found in Graph.hpp. The required method signature can be found in DFS.hpp, and your implementation will go in DFS.cpp. For this question, the vertex ID’s will be represented with integers, but you must follow the template class and use a generic T so that you can re-use DFS for later problems (and any other applications you may want to pursue outside of the class). We recommend an iterative DFS solution so that your method is less likely to be memory-constrained on larger graphs.

To run the tester:
make TestDFS
build/TestDFS testcases/input_file

Format for file input:
1 The first line gives the ID of the starting vertex for
1 2 DFS. Each subsequent line defines a directed graph edge
2 3 where the order is source vertex -> destination vertex.
2 4 <-- This is a directed edge from vertex 2 to vertex 4.)
3 4
4 5
6 5
Sample Output:
    build/TestDFS input_file
    Result of DFS: [ 1 2 3 4 5 ]

Again, you do not have to worry about the input / output as that will
be handled by the tester. Confirm that you have a working DFS
implementation by making your own test cases in the format described
above.

Question 2: Breadth-first Search [10 Points] :: BFS.cpp/hpp

Input:    a directed graph and a particular vertex ID
Output:   set of all vertices reachable from the given vertex ID

A vertex v is reachable from u if there is a directed path from u->v.
The set of vertices reachable from v includes v itself.

Next, you will implement a breadth-first search algorithm in a
similar fashion. Again, the required method signature can be found in
BFS.hpp and the implementation will be in BFS.cpp. As in Question 1,
we recommend an iterative BFS solution so that your method is less
likely to be memory-constrained on larger graphs.

To run the tester:
    make TestBFS
    build/TestBFS testcases/input_file

Format for file input:
    1       The first line gives the ID of the starting vertex for
    1 2     BFS. Each subsequent line defines a directed graph edge
    2 3     where the order is source vertex -> destination vertex.
    2 4    (<-- This is a directed edge from vertex 2 to vertex 4.)
    3 4
    4 5
    6 5

Sample Output:
    build/TestBFS input_file
    Result of BFS: [ 1 2 3 4 5 ]

Again, you do not have to worry about the input / output as that will
be handled by the tester. Confirm that you have a working BFS
implementation by making your own test cases in the format described above. Note that the output should be indistinguishable from your DFS algorithm in Question 1. That said, we will be checking your code to make sure you actually implemented BFS!

**Question 3: Topological Order [20 Points] :: TopOrder.cpp/hpp**

Input: a directed graph  
Output: a list of vertex IDs in a topological order  
(an empty list if such an ordering does not exist)

Given a DAG (directed acyclic graph) $G = (V,E)$ with $|V| = n$, a **topological ordering** is an assignment of the labels 1, 2, ..., $n$ to the vertices $v_i \in V$ such that every edge in $E$ is from a lower label to a higher label. If a graph $G$ is not a DAG, there will be no valid topological ordering of its vertices. [As mentioned in class, you can think of a topological ordering as a feasible order in which, say, tasks can be performed, given that there are precedence constraints between tasks. If there is a directed cycle in the graph, then no topological ordering exists: it is impossible to find a vertex that can receive the lowest label among the vertices in the cycle.]

For example, let’s take the lower div CSE class requirements:

```
(12)------------>(30)------------+
    ^           ^           |
    |           |           |
  +------>     |           | v
(11)------->(15L)--------->(100)
   |                             |
   |                             |
(20)--------->(21)-----+
```

A valid topological ordering is: [11, 12, 15L, 30, 20, 21, 100].  
Another equally valid ordering: [20, 11, 15L, 12, 21, 30, 100].

You’ll notice that as long as 11 is before 12 and 15L, 20 before 21, 30 and 21 before 100, etc. then the ordering is a topological ordering. Your TopOrder can output any ordering as long as it is a valid topological ordering.
The graph structure for this question also uses the graph definition from Graph.hpp, and relies on the TestTopOrder.cpp tester for input/output. In this case, the template variable used in the tester is a char. To keep the output of permutations reasonable, each vertex is represented by a distinct single alphanumeric character ranging from a-z, A-Z, or 0-9.

To run the tester:
   make TestTopOrder
   build/TestTopOrder testcases/testfile

Format for file input:
   H E      Each line defines a directed graph edge where
   G E      the order is source vertex -> destination vertex.
   E D      ← (this is a directed edge from vertex E to vertex D)
   E F
   D C
   C B
   B A
   F A

Sample Output:
   build/TestTopOrder testfile
   Result of Topological Ordering: [ G H E F D C B A ]

Or if the input graph is not a directed acyclic graph:
   build/TestTopOrder testfile
   Result of Topological Ordering: [ ]

Again, note that any ordering of the vertices is valid as long as it represents a correct topological ordering.

Question 4: Strongly Connected Components [20 Points] :: SCC.cpp/hpp

   Input: a directed graph
   Output: a list of sets - each set containing the vertices of a SCC

In this question, you will be implementing SCC-finding in a directed graph. As there are several excellent algorithms for finding SCCs,
you may choose to implement any algorithm as long as the algorithm has been proven to run in linear time. You are encouraged to create your own helper functions and use STL data structures as suitable.

We have included a tester through which you can run tests in a similar manner to that of the DFS tester, so make sure to use this to ensure correct, well-formatted output.

**To run the tester:**
make TestSCC
build/TestSCC testcases/input_file

**Format for file input:**
1 2  Each line defines a directed graph edge where
2 3  the order is source vertex -> destination vertex.
2 5  (this is a directed edge from vertex 2 to vertex 5)
2 6
3 4
3 7
4 3
4 8
5 1
5 6
6 7
7 6
8 7
8 4

**Sample Output:**
build/TestSCC input_file
SCC groupings:
6 7
3 4 8
1 2 5