Depth First Search and Topological Sort with Graphs
Lab #12

By
Michael Harden

CS 303 Algorithms and Data Structures
November 28th, 2021

1. Problem Specification

The goal of this lab is to implement a Graph data structure using an adjacency list and populate it with provided data. Then, implement a Depth First Search algorithm to calculate each path from a start node to each other node in the graph and test the code written. Finally, Implement a topological sorting method.

2. Program Design

This program contains two classes. First is the Graph class. This class implements the Graph data structure by means of an adjacency list. When creating an instance of this class, the graph can be designated as directed, or undirected. The methods provided allow the client to add nodes and/or connections between nodes, preform a depth first search over the graph, and when the graph is directed a topological ordering can be calculated.

The adjacency list is implemented using a hash table with integer keys. These keys are the number values of each node. the hash table values being stored at each key are a python set, containing the numerical value of all connecting nodes. Due to there being no need of maintaining order between nodes, the python set data structure is used. Using python sets allows for a constant look up time, on average.

The following methods are provided in the Graph class:

a) __init__(directed=False):

The init method is run when a new instance of the Graph class is created. It initializes the adjacency list as an empty hash table. This method also sets the directed property for the graph. Every graph's default is undirected. However, the directed parameter can be set to True to make it a directed graph.

b) add_node(node, neighbor):

The add_node method checks if the node provided is present in the adjacency list. If not, a key value pair is added to the adjacency list, with the key being the node, and the value being an empty set to store the neighboring nodes. After this check has been performed the neighbor is put in the node's neighbor set.

On an undirected graph this same process is then repeated with the node and neighbor swap places in the process. If the graph is directed, the same process is repeated, but the node is never put into the neighbor's neighbor set.

c) dfs(start):

The dfs method returns both the order in which all nodes are visited, and a list containing each path from the start node to each of the other nodes in the graph. Both these outputs are calculated to allow one dfs method to be used in calculating the paths as well as allowing the topological sorting method to calculate a topological ordering of the graph. The returned values are calculated by means of a helper method, _dfs_helper().

d) _dfs_helper(node, visited, paths=None, found=None):

_dfs_helper is a recursive helper method for both dfs() and topological_sort().

It starts by marking the node passed into the parameter as visited. Then the list of neighboring nodes to the node is retrieved from the adjacency list. Next a for loop steps through each neighbor and checks if they have been visited. If not, then the path of the neighbor is added to the list of paths. Its path is calculated by taking the path of node and adding ‘, V’ where v is the current neighboring node. Then _dfs_helper() is called recursively on that neighbor and the process continues until all nodes have been visited. Once all neighbors of a particular node have been visited, that node is added to the list of found nodes, signifying an end of a certain path. Then the list of paths and found nodes are returned. At the end of this recursive process a list of the paths from the start node to each other connected node is successfully calculated, and the list of nodes in found gives a reverse of the order in which ends of paths are found.

e) topological_sort():

Top sort requires that a graph be directed and acyclic. When running the method, first the directed property of the Graph class is checked. If it is an undirected graph, an error message is returned.

To perform the top sort, first an empty set is initialized to store all the nodes which have been visited. Then a list is created to store the topological order. Next all the nodes stored in the adjacency list are iterated through. If a node has not been visited, then dfs() is called starting with that node. The list, found, which is returned from dfs(), is saved into a variable found in the top sort and appended to the topOrder list. After all nodes have been visited, topOrder is reversed and it gives the order in which all nodes should be visited to verify all required previous nodes are visited first, the topological order.

The second class, Driver, provides methods to load a file into a graph, print each path from a start node to each of the other nodes in the graph, and calculate and print the topological ordering for an undirected acyclic graph.

a) load_file_to_graph (file_name, directed=False):
   This method takes a file name and/or file path as the parameter, opens and parses its contents into a graph. The state of the graph's directed property is the set with the directed parameter. The parser is designed to work with the format of files which are provided on canvas for this lab. The graph is saved into an instance variable, graph, for use by the Driver class.

b) dfs(start, npath=100):
   The dfs method prints each path from the start node to each of the other nodes in the graph. These paths are calculated using Graph.dfs algorithm. Start is the node which the searching originates from, and npath is the number of paths to be printed from the list of paths returned from Graph.dfs(). If None is passed for npaths, or if the number of paths exceeds the total paths calculated, all the paths are printed. To print 0 paths, 0 must be pasted. The default number of paths to print is the first 100.

c) test_dfs():
   This method creates the graph given on the lab11 doc and prints its adjacency list, and all the paths calculated by dfs, verifying the Graph class is working correctly.

3. Testing

To test the Graph class, the graph from lab11 document, provided on canvas, was created. Then the adjacency list is printed, along with the paths starting at 0 to all nodes. The first step ensures the graph adjacency list is being formed correctly, and the latter step ensures depth first search is working properly

When testing in this manor the following output was printed to the command line

```
1: {2, 5}
2: {1, 3, 4, 5}
5: {1, 2, 4}
3: {2, 4}
4: {2, 3, 5}
1: 1
2: 1 , 2
3: 1 , 2 , 3
4: 1 , 2 , 3 , 4
5: 1 , 2 , 3 , 4 , 5
```

Checking this output alongside the graph provided, it verifies the Graph class is working correctly

In the lab 12 instructions, it states to use tinyDG.txt file to test the topological sort. However, although this graph is undirected, there are many cycles such as <4, 5, 4> and <0, 2, 7, 3, 6, 0> Making it impossible to get a topological ordering. A topological ordering requires no cycles because if 1 requires 2 and 2 requires 1, neither can ever come first. Due to this, a separate file was created titled "tinyAcDG.txt" this file contains no cycle. It can be found in the assignment12 zip file.

When testing Topological sort on this file the ordering calculated is [1, 4, 2, 3, 7, 5, 6, 8]. This is correct as each of the nodes' dependencies come prior to the corresponding node. 7 is dependent on 3 and 4, both of which come before 7, ect.

4. Analysis and Conclusion

In conclusion when storing information in a graph data structure an adjacency list allows for much information to be stored without wasting space, while maintaining quick searching time. Furthermore, based on how the nodes are being store in the list, look up time for a particular node can be very fast. In the case of using a hash table, when using a hashing function with uniform density, look up times are close to constant time.

Depth First Search Big-O time is O(V+E). So, the speed at which a depth first searches can be performed is directly proportional to the size of the graph. This lends dfs to be a great algorithm for calculating many properties in a graph. For instance the degree of connectedness of a graph, or as seen in this lab depths first search can be used for calculating a Topological ordering.

Top sort can be us calculate the order in which programs dependencies must be run to prevent crashes or used in scheduling problems. Top sort does not give a traditional alpha-numeric ordering, but rather provides one of possibly many orderings in which events must happen to allow for all prior dependencies to be met for latter events to follow.

5. Credits

- The example lab report provided on canvas was used as a template for this lab report.
- For research on top sort: https://youtu.be/eL-KzMXSXXI