Analyzing Breadth First Search with Graphs
Lab #11

By
Michael Harden

CS 303 Algorithms and Data Structures
November 16th, 2021

1. Problem Specification

The goal of this lab is to implement an undirected Graph data structure as an adjacency list and populate it with provided data. Then, implement a Breadth First Search algorithm to calculate, for each node, the path from a start node to each of other nodes in the graph and test the code written. And finally, to analyze the time of Breadth First Search

2. Program Design

This program contains two classes. First is the Graph class. This class implements the Adjacency Graph data structure. It provides methods to add a node and/or a connection from that node to another node in the graph, and to search the graph with Breadth First Search. The second class is the Driver class. This class provides methods to parse a csv file into a graph, preform Breadth First Search over that graph, time BFS and test the implementation of the graph and BFS algorithm

The adjacency list is implemented using a hash table with integer keys, which are the numbers of each node, and the value being stored alongside each key is a python set, containing the integers representing connecting nodes. The hash table and python set data structures are used since it is not necessary for order to be maintained. Using hash tables and sets allows for an average O(1) look up time, every time a node is searched for.

The following methods are provided in the Graph class:

a) __init__(size):
   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

b) add_node(node, neighbor):
   The add node function puts the node provided into the adjacency list, along with a python set which will contain the connecting nodes. If the node to be added is already present in the graph, then no node is put into the graph, and the new connection to the neighbor node is added to the set of the neighbors for the present node that was passed into the parameter, node.

   Since the graph is undirected the same process happens again with the node and neighbor's position in the process reversed.

c) bfs(start):
   The bfs method visits every node in a breath first manor. The start node is first visited, followed by all its neighbors, then the neighbor's neighbors are visited. This process is continued until all nodes in the graph have been visited. A queue is used for processing each node, as the FIFO property allows for a n+1 level neighbor node to be saved and visited once all n

level neighbors have been visited. When a node is visited, each of its neighbors are queued to be visited later.

Once they have been queued, that node is added to a set which keeps track of all the nodes which have been queued. Before any node is queued, this set is checked, and if that node has already been queued it will not be requeued. This prevents cycles in the searching.

To calculate all the paths, there is another hash table that stores a node as a key, and the path to that node as the value, saved as a string. When a node is visited, the path to its previous node is taken from the table, and the current node is then appended to the end of that string. This process starts with the start node. Its path is saved as only itself.

The following methods are provided in the Driver class:

a) load_file_to_graph (file_name):
   This method takes a file name or path in as the parameter, opens and parses its contents into an undirected graph. The parser is designed to work with the types of files which are provided on canvas for this lab. The graph is saved into an instance variable for use by the class

b) time_bfs(start, npath=100):
   The time_bfs method times the Graph.bfs algorithm. It calls the Graph.bfs() function. Start is the node that the searching originates from, and npath is the number of paths to be printed from the list of paths returned from Graph.bfs(). If None is passed for npaths, or if the number of paths exceeds the total paths, all the paths are printed. To print 0 paths, 0 must be used instead. This defaults to the first 100 paths. The time taken to preform bfs is calculated my saving the time prior to running bfs and then comparing it to the time after completing bfs. That time is printed to the command line. This same process happens for both files.

c) test_bfs():
   This method creates the graph given on the lab11 doc and prints all its adjacency list, and paths, 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 breadth 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
5: 1 -> 5
3: 1 -> 2 -> 3
4: 1 -> 2 -> 4
```

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

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 wasted space, as well as 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, as explored in lab 9, when using a hashing function with uniform density, look up times are close to constant time.

To preform Breadth First Search the Medium size graph, the time taken is 5.13E-4 seconds. To preform BFS on the large size graph, the time taken is 1.14E+1 seconds, with a Big-O time complexity of O(V+E)

Breadth First Search is useful when attempting to find 1st through nth degree neighboring nodes, as well as calculating the most direct path when all connections have the same weight or trying to calculate what degree of connection two nodes have in an efficient manor.

5. Credits

- The example lab report provided on canvas was used as a template for this lab report.