Olivia Caulfield
Devarakonda
CSC 301
October 20, 2023

Homework 5

Code Description:

I. I chose to write my project in java.

II. Depending on the inputs, the code will either calculate the two-paths between two vertices, or return an argument error.

III. After reading the edge input file into an array of linked lists which is the size of the number of vertices (*input_list*) and initializing the output adjacency list as an array of hashmaps (*output_list*), I call my BFS method on each linked list in my array.

IV. My BFS algorithm is defined in the function *BFS* which accepts three parameters, an array of linked lists, *input_list*, an array of hashmaps, *output_list*, and the index of the array you are completing BFS on, *start*.

   A. First, a queue is initialized in order to hold the next nodes for processing, and the start node is added to the queue.

   B. Then three variables are initialized:

      1. *depth* is the level of the BFS tree that you are currently at, initialized to 0 for the top of the tree.
      2. *curr_nodes* is the number of nodes in the current level, initialized to 1 since only the start node is in the queue.
      3. *next_nodes* is for storing the number of nodes that will be in the next level of the tree, initialized to 0 since the number of nodes at the next level will be unknown until iterating through the current level's neighbors.

   C. Next, the queue is looped through on two conditions:

      1. *The queue has elements:* This will stop the loop if there are no more nodes to explore.
      2. *The depth is less than or equal to 2:* this is because the code is looking for two-paths, so it is unnecessary to explore past a depth of two. The first depth finds nodes one step away from the start node, therefore, the second depth will find the neighbors of the first depth's nodes, or in other words, nodes that are two steps away from the start node.

   D. Inside the loop the current node is set equal to the node taken off the queue in the variable *current*.

      1. If the node being processed is at a depth of 2, its value and its corresponding count will be added to the hashmap based on one of two conditions:

a) If the node does not exist in the hashmap (*output_list[start-1].get(current+1) == null)* then the node will be added to the hashmap and its count will be initialized to 1.
b) Otherwise, the node already exists in the hashmap and its count will be incremented by 1.
c) Therefore, the key of the hashmap entry is equal to the current node value and the value of the hashmap entry is the total number of two-paths between the start node and the current node.

2. In doing this, a weighted list is created which stores how many two-paths exist between the start node and the node being currently processed.

E. Next, the value of *curr_nodes* will be decreased by 1 since the node has been removed from the queue and is being processed.
F. Then, each of the current node's neighbors will be added to the queue and the number of nodes in the next depth (*next_nodes*) will be incremented by 1 each time.
G. Finally, if *curr_nodes* is equal to 0, that means that all of the nodes at the current tree level have been processed, thus the counts (*curr_nodes* and *next_nodes*) should be restarted and the depth should be incremented by 1.

V. Finally, the *get_reccomendations* method is called in order to process the information in the hashmap and determine the best follower recommendations. *get_reccomendations* accepts two arguments. The first is the weighted adjacency list *list_array* and the second is the input adjacency array *input*.
A. For each node in the *list_array*, elements in its hashmap that are equivalent to itself or a node that the current element already follows in the *input* are removed.
B. Then each entry in the hashmap is looped through in order to find which entry has the highest value (number of two-paths), while breaking ties using the lower key.
C. Then, the recommendation is printed to the user.

Complexity Analysis:

After looking at the code in depth in the description above, I will now describe it from a higher level in order to compute an upper bound on the complexity of the algorithm in terms of the number of edges (m), nodes/vertices (n), and the maximum out-degree of any vertex (d)

- First, in the BFS method any node can have at most d outward edges. Thus for each start node, the maximum number of neighbors to explore to find one-paths will be d. Therefore, the time-complexity of finding one-paths for one start node will be $O(d)$.
- Furthermore, to find two-paths, each of the start node's neighbor nodes (which are at maximum d) will need their neighbors explored (and each neighbor can have at maximum d neighbors). Therefore, the time-complexity of finding the two-paths will be $d*d \rightarrow O(d^2)$.

- Finally, in the main method, the BFS method is called one time for each of the nodes. Therefore, in total BFS will be called n times.
- Since BFS has a complexity of $O(d^2)$ and it is called n times, the upper bound of the total complexity for computing two paths stored in the adjacency list is $O(nd^2)$.
- Thus, when $d < n$, the time complexity of this algorithm is less than $O(n^3)$.

Recommendations:

Using the *list_reccomendation* code, I computed the output recommendations for the 1024 edge list. Below are some examples of recommendations for the word list:

1. 487 → 473: reasoning → experiment
    a. 487 has 7 neighbor nodes:
    25, 471, 488, 489, 491, 548, 735
    b. Of the 7 neighbors, two of them are neighbors with 473:
    471, 489
    c. Thus the intermediate words that correspond with this recommendation are:
    471: inquiry
    489: demonstration

    Overall, this recommendation seems meaningful because "reasoning" could be related to "experiment" through a change from a theoretical process (reasoning) to a hands-on process (experiment).
2. 465 → 545: curiosity → secret
    a. 465 has 4 neighbor nodes:
    466, 471, 519, 544
    b. Of the 4 neighbors, two of them are neighbors with 545:
    471, 544
    c. Thus the intermediate words that correspond with this recommendation are:
    471: inquiry
    544: news

    Overall, this recommendation seems meaningful because "curiosity" could be related to "secret" in the way that one's expressed interest to know something (curiosity) is due to the fact that it is concealed (secret).
3. 416 → 412: resonance → silence
    a. 416 has 3 neighbor nodes:
    b. Of the 3 neighbors, one of them is neighbors with 412:
    413
    c. Thus, the intermediate word that corresponds with this recommendation is:
    413: faintness

    Overall, this recommendation seems meaningful because "resonance" could be related to "silence" in the way that the elimination of a sound (resonance) results in no more sound (silence).

Time Comparison:
Below is a table containing the times it took in nanoseconds for the list and adjacency approach to calculate two-paths for different sized inputs.

Time to Calculate Two-Paths

Size (lines)	**List Time (ns)**	**Matrix Time (ns)**	**Percent Decrease**
16	1,093,270	2,606,749	58.06%
1024	39,756,872	267,467,466,447	99.99%

As evident above, in using the list method there was a 58% decrease in time needed to calculate two-paths as compared with the matrix method for an input of size 16. Similarly, for an input of size 1024, there was a decrease of 99.99% time required to calculate the two paths using the list method as compared to the matrix method. Therefore, it is clear that the list approach is faster.

The list approach is faster for two main reasons; the input graphs are sparse and the time complexity of the matrix approach is $O(n^3)$ while the time complexity of the list approach is $O(nd^2)$, as proved above. Looking at the matrix approach, it is evident that regardless of the number of edges in the graph, the method still has to complete a multiplication for each possible combination. Therefore, for a graph of n vertices, these multiplications have a cost of n^2 making it much more time intensive to compute a two-path ($O(n^3)$). On the other hand, in the list approach, each vertex is bound to have an out-degree of d which is less than n, and thus much less than n^2. Thus, for the sparse input graphs given, $nd^2 << n^3$.