Nicole Telesz
Amanpreet Kapoor
COP3530
09 October 2020

Graphs and PageRank Documentation

Describe the data structure you used to implement the graph and why?

I chose to use a map to implement my graph. The associative property between the first and second elements (key and value) was vital for representing edges between vertices. My map used a string of the vertex as the key and a vector of string/double pairs as the value. The pairs, a string for a url and a double for the matrix value, were put in a vector to represent the multiple vertices to which the vertex in the key pointed. Std::map is implemented as a Red-Black tree behind the scenes and thus has relatively efficient operations in terms of time complexity, which is elaborated upon in the following questions.

What is the computational complexity of each method in your implementation? Reflect for each scenario: Best, Worst, and Average.

Operations involving any of my maps will have time complexities reliant upon the fact that std::map is implemented as a Red-Black tree behind the scenes.

method	**description of implementation**	**best**	**worst**	**average**
void insertEdge(string from, string to)	pushes back “to” into adjListIntermediate at “from”, adds “to” to adjListIntermediate if not already present, increases totNumEdges count	accessing in map is O(1) where variable relates to number of vertices in graph, push_back is constant O(1) where variable relates to number of elements in vector, incrementing totNumEdges in constant O(1), within if statement find() is O(1) and end() is O(1) where variable relates to number of vertices in graph, following insertion is O(log N) like previously stated overall: O((1 + 1) + 1 + ((1 + 1) * logN)) = O(logN)	accessing in map is O(log N) where N is number of vertices in graph, push_back is constant O(1) where variable relates to number of elements in vector, incrementing totNumEdges in constant O(1), within if statement find() is O(log N) and end() is O(1) where variable relates to number of vertices in graph, following insertion is O(log N) like previously stated overall: O((logN + 1) + 1 + ((logN + 1) * logN)) = O((logN)^2)	accessing in map is O(log N) where N is number of vertices in graph, push_back is constant O(1) where variable relates to number of elements in vector, incrementing totNumEdges in constant O(1), within if statement find() is O(log N) and end() is O(1) where variable relates to number of vertices in graph, following insertion is O(log N) like previously stated overall: O((logN + 1) + 1 + ((logN + 1) * logN)) = O((logN)^2)

method	**description of implementation**	**best**	**worst**	**average**
void makePageRank()	iterates through each vector in adjListIntermediate, determines matrix values (1/outdeg) for each value depending on size	accessing in map is O(1) where variable relates to number of vertices in graph, if statement comparing size and constant is O(1) with same reference variable, push_back is O(1) with same reference variable, above sequence happens either once or M times where M is size of vector associated with current adjListIntermediate pointer, whole process happens as many times as N elements in adjListIntermediate overall: O((1 * N) * (1 * 1)) = O(N)	accessing in map is O(log N) where N is number of vertices in graph, if statement comparing size and constant is O(1) with same reference variable, push_back is O(1) with same reference variable, above sequence happens either once or M times where M is size of vector associated with current adjListIntermediate pointer, whole process happens as many times as N elements in adjListIntermediate overall: O((logN * N) * (M * 1)) = O(logN*NM)	accessing in map is O(log N) where N is number of vertices in graph, if statement comparing size and constant is O(1) with same reference variable, push_back is O(1) with same reference variable, above sequence happens either once or M times where M is size of vector associated with current adjListIntermediate pointer, whole process happens as many times as N elements in adjListIntermediate overall: O((logN * N) * (M * 1)) = O(logN*NM)
void powerIter(int pow)	determines initial r and pushes back into powerIters vector, if pow is 1 then pushes url and r into finalOutput, else goes through loop of doing matrix multiplication and copying value into powerIters, pushes url and r into finalOutput	r calculation is constant time O(1) where variable relates to number vertices in graph, for loop of constant O(1) pushbacks occurs N times where N is number of vertices in adjListIntermediate, best case pow is 1 and loop of N pushbacks value of powerIter which is O(1) access for same reference variable and increments i at constant time overall: O(1 + N + N) = O(N)	r calculation is constant time O(1) where variable relates to number vertices in graph, for loop of constant O(1) pushbacks occurs N times where N is number of vertices in adjListIntermediate, else has loop that iterates P times where P is pow - 1, checking if empty is O(1) and clear is O(N), following for nested for loop iterates from outermost to innermost N, N, M times where M is size of current dereferenced pageRanks iterator,	r calculation is constant time O(1) where variable relates to number vertices in graph, for loop of constant O(1) pushbacks occurs N times where N is number of vertices in adjListIntermediate, else has loop that iterates P times where P is pow - 1, checking if empty is O(1) and clear is O(N), following for nested for loop iterates from outermost to innermost N, N, M times where M is size of current dereferenced pageRanks iterator,

method	description of implementation	best	worst	average
(cont.)			multiple constant time incrementations, accessing map information is O(logN), accessing vector information is O(1) with variable in relation to size of vector, copying values with access of O(1) N times, transfer iterates N times with insertion of O(log N) and constant time incrementations of i overall: O(1 + N + (P * (N * N * M * logN) + N) + (N * logN)) = O(N + PN^2MlogN + NlogN) = O(N(PNMlogN+log N)) = O(NlogN(PNM)) = O(N^2*logN*PM)	multiple constant time incrementations, accessing map information is O(logN), accessing vector information is O(1) with variable in relation to size of vector, copying values with access of O(1) N times, transfer iterates N times with insertion of O(log N) and constant time incrementations of i overall: O(1 + N + (P * (N * N * M * logN) + N) + (N * logN)) = O(N + PN^2MlogN + NlogN) = O(N(PNMlogN+log N)) = O(NlogN(PNM)) = O(N^2*logN*PM)
void printGraph()	creates iterator to traverse finalOutput map and prints first and second associated with *itr	when creating iterator begin() and end() have constant time complexity O(1) where variable relates to size of map, itr++ has constant time complexity O(1), to go through entire map it must traverse through all N elements and thus has time complexity of O(N), printing dereferenced values is constant time complexity 0(1) where variable relates to number of elements in map overall: O(N*1) = O(N)	when creating iterator begin() and end() have constant time complexity O(1) where variable relates to size of map, itr++ has constant time complexity O(1), to go through entire map it must traverse through all N elements and thus has time complexity of O(N), printing dereferenced values is constant time complexity 0(1) where variable relates to number of elements in map overall: O(N*1) = O(N)	when creating iterator begin() and end() have constant time complexity O(1) where variable relates to size of map, itr++ has constant time complexity O(1), to go through entire map it must traverse through all N elements and thus has time complexity of O(N), printing dereferenced values is constant time complexity 0(1) where variable relates to number of elements in map overall: O(N*1) = O(N)

What is the computational complexity of your main method in your implementation? Reflect for each scenario: Best, Worst, and Average.

The computational complexity of my main method depends on the calculated time complexities of each of the functions that it references. My main function takes input from the user by line and inserts edges into the graph that many times. After this, it calls makePageRank(), powerIter(int), and printGraph().

	best	**worst**	**average**
main	for loop iterates L times where L is number of lines and inserts edge each time which is O(log N) where N is number of vertices, following functions have time complexities as stated in table overall: O(LlogN + N + N + N) = O(LlogN + N)	for loop iterates L times where L is number of lines and inserts edge each time which is O(log N) where N is number of vertices, following functions have time complexities as stated in table overall: O(L(logN)^2 +NMlogN + N^2*logN*PM + N)	for loop iterates L times where L is number of lines and inserts edge each time which is O(log N) where N is number of vertices, following functions have time complexities as stated in table overall: O(L(logN)^2 +NMlogN + N^2*logN*PM + N)

What did you learn from this assignment and what would you do differently if you had to start over?

From this project, I learned how to work with adjacency lists and how to use them to represent matrices. I also learned how to do matrix multiplication in order to do the power iterations. One of the most important things I learned from this project was how to be more comfortable using maps and the power you can have with your choices for their keys and values.

If I had to start over, I would try to focus on optimizing my functions and methods. I feel like a lot of my code utilizes multiple levels of nesting, which is not very efficient in regards to time complexity. I would also try refine my methodology for calculating page rank through power iterations. My solution seems more convoluted than it needs to be.