Yash Hegde

Professor Kapoor

Data Structures & Algorithms

19 July 2021

P2 - Documentation

1. **Q: Describe the data structure you used to implement the graph and why?**

A: The data structure that I used to implement the graph was the map, specifically its two variations - the unordered\_map and map. The unordered\_map was responsible for storing a website as a key, with the value being a vector of websites that the key website is directed towards. Alternatively, the map was used for storing a pair of double data types that represented a website’s page ranking after *i* iterations. I decided to use a map where ordering of the output was important, due to its rather low time complexity, being O(*log n*) (This is O(*log n*) because a map’s implementation is essentially a red-black tree). Where ordering was not important, however, an unordered\_map was used, due to its lower time complexity than that of a map. In many instances, an unordered\_map’s runtime would amount to O(1 + *K*), with *K* being the amount of time it would take the hashing function involved to find an index in the unordered\_map.

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

A:

*Insert Method -*

*Worst Case:* O(*log n + K*), where *n* is the height of the red-black tree that allows for unordered\_map (m) functionality, and *K* is the amount of time it would take the hashing function involved to find an index in the map (mFinal). This would represent a case where many website names are already placed in the maps, and the last name is to be placed.

*Average Case:* O(*log n + K*), where *n* is the height of the red-black tree that allows for unordered\_map (m) functionality, and *K* is the amount of time it would take the hashing function involved to find an index in the map (mFinal). This would represent a case where some arbitrary number of website names have been inserted, and a name is being inserted before some other insertions are made.

*Best Case:* O(*K*), where *K* is the amount of time it would take the hashing function involved to find an index in the map (mFinal). This represents a case when the value being inserted is a duplicate name, and the number of vertices in the graph would not affect the time complexity.

*isEdge Method -*

*Worst Case:* O(*n*), where *n* is the size of the vector of strings that the key value of unordered\_map (m) is directed towards. The loop in the method would have to iterate through the number of indices, equivalent to the size *n* to check if a website is connected to the passed in parameter, when the website is not in the vector.

*Average Case:* O(*i*), where *i* is the number of strings that the function iterates through to find the website that is equal to the passed in parameter (to). The loop in the method would have to iterate through any arbitrary number of indices to find a website connected to the passed in parameter.

*Best Case:* O(1), which represents a case where the value being checked does not exist, or is being checked with an empty vector in the unordered\_map (m).

*getAdjacent Method -*

*Worst Case:* O(*log n + rp*), where *n* is the height of the red-black tree that allows for unordered\_map (m) functionality in the find function, *r* is the size of the unordered\_map (m), and *p* is the size of the vector of strings that the key value of unordered\_map (m) is directed towards. The function implements the find function of the map, and iterates through the entire map (mFinal) with the isEdge function as a condition, regardless of whether the website is or is not found based on the passed in parameter (to).

*Average Case:* O(*log n + ri*), where *n* is the height of the red-black tree that allows for unordered\_map (m) functionality in the find function, *r* is the size of the unordered\_map (m), and *i* is the number of strings that the function iterates through to find the website that is equal to the passed in parameter (to) in the isEdge function. The function implements the find function of the map, and iterates through the entire map (mFinal) with the isEdge function as a condition, regardless of whether the website is or is not found based on the passed in parameter (to).

*Best Case:* O(*log n + r*), where *n* is the height of the red-black tree that allows for unordered\_map (m) functionality in the find function, *r* is the size of the unordered\_map (m). The function implements the find function of the map, and iterates through the entire map (mFinal) with the isEdge function as a condition, regardless of whether the website is or is not found based on the passed in parameter (to). We cannot assume the red-black tree is empty, or the size of the map is empty in the best case, due to the way input is taken in the main method.

*Pagerank Method -*

*Worst Case:* O(*pij*), where *p* is the number of power iterations, *i* is the size of the map (mFinal), and *j* is the size of the vector of all adjacent website names of some website in map (mFinal). This function has an outer loop that iterates to *p*, and two inner loops that iterate to *i* and *j*. The *j*, in particular, is a result of an iterator passed into the getAdjacent function (*i* is always going to be greater than or equal to the size of unordered\_map (m), so getAdjacent’s time complexity would be neglected in the final calculation).

*Average Case:* O(*pij*), where *p* is the number of power iterations, *i* is the size of the map (mFinal), and *j* is the size of the vector of all adjacent website names of some website in map (mFinal). This function has an outer loop that iterates to *p*, and two inner loops that iterate to *i* and *j*. The *j*, in particular, is a result of an iterator passed into the getAdjacent function (*i* is always going to be greater than or equal to the size of unordered\_map (m), so getAdjacent’s time complexity would be neglected in the final calculation).

*Best Case:* O(*pi*), where *p* is the number of power iterations, and *i* is the size of the map (mFinal). This case would be similar to the above two, except for the fact that the inner loop that executes until *j* would not enter the if statement if there are not enough power iterations to justify more calculations to be made.

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

A:

*Worst Case:* O(*Llog n + LK + pij*), in which *L* is the number of lines of input, and *log n + K* being the worst case complexity of the insert method in the loop (T) in main that goes from 0 to *L-*1. Additionally, *pij* is the worst case of the pagerank method that coincides with loop (T).

*Average Case:* O(*Llog n + LK + pij*), in which *L* is the number of lines of input, and *log n + K* being the average case complexity of the insert method in the loop (T) in main that goes from 0 to *L-*1. Additionally, *pij* is the average case of the pagerank method that coincides with loop (T).

*Best Case:* O(*LK + pi*), in which *L* is the number of lines of input, and *K* being the best case complexity of the insert method in the loop (T) in main that goes from 0 to *L-*1. Additionally, *pi* is the best case of the pagerank method that coincides with loop (T).

1. **Q: What did you learn from this assignment and what would you do differently if you had to start over?**

A: From this assignment, I learned the importance of being able to place different containers within another type of container (for example, putting a vector in a map) to be able to increase any particular data structure’s functionality. Additionally, I realized the importance of labelling variables when discussing the time complexity of a program, as larger programs have many loops and iterations involved. If I were to start over, I would make sure to use a catch framework with my program for better debugging routines, rather than relying on a “run” button, or something similar in function.