Simplified PageRank Report

Hector Caban Barreto

Data Structure

Overview

I chose to represent the adjacency list data structure with an std::unordered_map with std::string being the key and std::unordered_map being the value. The structure of the nested std::unordered_map was std::string as the key and double being the value. So overall the structure of the adjacency list itself was `std::unordered_map<std::string, std::unordered_map<std::string, double>>`

Breakdown

Since this adjacency list deals with strings to identify vertices, there has to be a mechanism to hash string values to their respective list of adjacent vertices thus using a map was the obvious choice. I chose to prioritize access time over the ordering of vertices thus that is way std::unordered_map was chosen giving `std::unordered_map<std::string, containter>`, each string (url) maps to a container the contains their adjacent vertices.

Now we must decide what kind of container will be used to represent the list of adjacent vertices that each vertex contains. One choice could be `std::vector<std::pair<std::string, value>>` which is sufficient enough to describe the list of adjacent vertices but a more future proof option would be `std::unordered_map<std::string, value>` as it will provide a faster average access time compared to the previous option though worst-case access time will be similar (will be elaborated on futher in the next section).

Now that we have developed a structure to represent the adjacency list that maps strings to their respective list of adjacent vertices we must now the decide the value contained within those lists which will just be a `double` thus the final data structure is `std::unordered_map<std::string, std::unordered_map<std::string, double>>`.

Time Complexity of Methods

Before I go into the analysis I must first list the average-case and worst-case time complexities of each of the std::unordered_map operations that will be used and are referenced throughout the analysis. Also I did the average-case calculations not knowing only worst-case was required but I have left them in since the results are interesting.

- **bracket operator:** `std::unordered_map<key, value>::operator[]`
  - $\Theta(1)$

  - O(n)
- **find:** `std::unordered_map<key, value>::find`
  - $\Theta(1)$
  - O(n)
- **size:** `std::unordered_map<key, value>::size`
  - $\Theta(1)$
  - O(1)

addEdge - $\Theta(1)$ / O(V)

In the beginning of the function, when checking if adjList[from][to] exists, adjList.find() is called twice and the bracket operator is used twice, giving average-case: T(V,E) = 4 and worst-case: T(V,E) = 4V. Then the bracket operator is used twice sequentially with adjList[from][to] thus average-case: T(V,E) = 4 + 2 and worst-case: T(V,E) = 4V + 2V. Find and the bracket operator are used again sequentially to check if adjList[to] exists and to create it and following that adjList[from] is accessed and adjList[from].size() is called thus so far bringing the average-case time complexity to T(V,E) = 4 + 2 + 4 and the worst-case to T(V,E) = 4V + 2V + 3V + 1. In the range-based for loop, adjList[from] is called and iterated through. In the average-case, the call to adjList[from] is constant and iterating through every element of adjList[from] will be $\Theta(1)$ since the graph would be sparse in the average-case and the worst-case adjList[from] is accessed and the number of elements in adjList[from] would be O(V) since the graph is dense in the worst-case. Thus bringing the final average-case to T(V,E) = 4 + 2 + 4 + 2 ~ $\Theta(1)$ and worst-case to T(V,E) = 4V + 2V + 3V + 1 + 2V ~ O(V).

getPageRank - $\Theta(p(V + E))$ / $O(pV^3)$

For this analysis I will skip to the section with the triple nested for loops since the analysis of the code before the for loops gives a similar result to addEdge, which is overshadowed by the for loops. It must be noted that as previously mentioned, the graph is not guaranteed to be sparse and will be assumed to be dense in the worst-case and will be assumed to be sparse ($\Theta(V + E)$) in the average-case. The first for loop executes p times, the second for loop executes once for each vertex in the graph, V times, and the number of executions of the last loop depends on the-case. In the average-case each vertex is visited in the second loop and each edge of each vertex is visited in the third loop thus those two loops combined (ignoring the constant time access to rankVector/Prime) gives $\Theta(V + E)$. In the worst-case, each vertex would have O(V) adjacent vertices so the third loop would execute O(V) times and each access to rankVector[row.first] and rankVectorPrime[column.first] would be O(V). With all that considered, the average-case time complexity would be T(V,E) = p * (V + E) ~ $\Theta(p(V + E))$ and the worst-case time complexity would be T(V,E) = p * V * V * 2V ~ $O(pV^3)$.

main - $\Theta(pV + pE + n)$ / $O(pV^3 + nV)$

The first for loop in the main function will iterate n (no_of_lines) times and `addEdge` will be called on each iteration plus `getPageRank` is called immediately after the for loop, thus the average-case $T(V,E) = n + p(V + E)$ and in the worst-case $T(V,E) = nV + pV^3$. The final for loop will iterate V times thus the average-case time complexity is $T(V,E) = n + p(V + E) + V \sim \Theta(pV + pE + n)$ and the worst-case time complexity is $T(V,E) = nV + pV^3 + V \sim O(nV + pV^3)$. Note that p, n >= 1 thus pV/nV will always grow as fast or faster than V.

Reflection

What I Learned

Before implementing the adjacency list I was skeptical of the benefits of implementing a graph as an adjacency list as most of the computation would deal with computing matrix multiplications so doing an adjacency matrix seemed like a no brainer to me, falsely believing that the overhead required perform matrix multiplications using an adjacency list would outweigh the wasted space of an adjacency matrix implementation. Though as I went through implementing the matrix multiplication algorithm with the adjacency list I found that there was essentially no extra overhead.

What I Would Do Differently

Although it is not standard, making the indexing format for adjList to be adjList[to][from] instead of adjList[from][to] would have made my life so much easier when it came time to implement the matrix multiplication algorithm since the PageRank adjacency matrices are indexed adjMatrix[to][from]. The main reason I did not decide to revamp my entire project to accommodate that was, as mentioned earlier, adjList[from][to] is a more standard indexing format which would help with portability if I do decide to use the data structure in a later project (like project 3).