COP 3530 - Project 2

Aiden Zepp

2023-11-14

Data Structure Analysis

The `Graph` class represents an adjacency-list implementation of a graph data structure. To hold the data, the `std::map` and `std::vector` data structure were chosen in the form of `std::map<std::string, std::vector<std::string>>`. The `std::map` suits an adjacency-list graph's need for storing data pertaining to a specific value. Likewise, the `std::vector` allows for simple and fast insertions while also meeting the criteria of an adjacency list. Both the `std::map` and `std::vector` were mostly chosen for their $O(1)$ insertion time complexity. Although the `std::vector` does not strictly have a $O(1)$ insertion time complexity, the data structure does have an *amortized* insertion time complexity of $O(1)$.

Because the project does not specify whether multiple edges are allowed between any two nodes (as might happen in reality, where a website can contain multiple links to the same webpage) and because the edges between nodes have no weight, a `std::vector` is more appropriate than using, say, another `std::map`. If the `Graph` additionally contained another data field such as `std::map<std::string, int>`, said data field would allow for constant time access of the stored `std::vector`.

Lastly, to avoid the annoyances of $O(n^2)$ time complexities for calculating the nodes that point to a given node (or, vice versa, for calculating the nodes that a given node points to), both directions of adjacency lists are stored in the `Graph`. Understandably, this is highly nonoptimal as the storage requirement doubles by doing so. However, I was unable to determine a better solution to mitigate this issue. For example, instead of storing a `std::vector<std::string>` the map could instead store a `std::vector<std::pair<std::string, double>>` where the `double` represents $\frac{1}{outdegree(node)}$. Yet this doesn't seem possible as there is no way of knowing during the insertion process what the final out-degree of any given node is until all nodes have been inserted. Therefore, having two data fields seems to be the most straightforward solution while still adhering to the adjacency-list implementation. This trade off, in my opinion, is acceptable as the *Page-Rank* algorithm is the focus of this assignment, not the amount of space the structure uses.

Time Complexity Analysis of `main`

The `main` function simply calls `Clap().Run()`. The `Clap()` constructor method is $O(1)$. The `Run()` method requires parsing l number of lines provided by the user and subsequently running the `Graph::Insert` and `Graph::PageRank` methods. Since the insertion must be done l times and the page-rank is calculated only once, the time complexity of `Clap::Run` and thus the `main` function is: $O(l) * O(1) + O(p * n^2) = O(l) + O(p * n^2) = O(l + p * n^2) \sim O(p * n^2)$. Therefore, the time complexity of `main` is bounded by the time complexity of `Graph::PageRank`.

Time Complexity Analysis of `Graph`

The following are time complexity analyses of the `Graph` class's functions. Please note that in order to not surpass the project assignment's 3 page limit for this document, only the time complexities of the `Graph` class's major functions are listed. All other functions not listed below are likely either menial or have a time complexity of $O(1)$.

For reference, recall that the `Graph` class has the underlying `Data from` and `Data into` data fields containing the graph's data. For reference of the underlying type definitions provided by `Graph`, please refer to the code-base.

Analysis

Public Methods

```
Page PageRank(unsigned int power) const;
```

The `PageRank` method requires the use of both versions of `GetPage` (see their respective analyses below). Since `GetPage()` has a time complexity of $O(n)$ and `GetPage(...)` has a time complexity of $O(p * n^2)$, this function's total time complexity is $O(n) + O(p * n^2) = O(n + p * n^2) \sim O(p * n^2)$.

```
void Insert(const Node& origin, const Node& target);
```

The `Insert` method requires the use of its private-method counterpart. Since insertion operations into a `std::map` have a time complexity of $O(1)$, this function has a time complexity of $O(1)$.

Private Methods

```
List GetList(const Node& node, const Flow& flow) const;
```

The `GetList` method merely accesses the respective data field and returns the `List` associated with the provided `node`. For example, `GetList(node, Flow::From)` would return `from.at(node)` and `GetList(node, Flow::Into)` would return `into.at(node)`. Since access operations on a `std::map` have an $O(1)$ time complexity, this function has an $O(1)$ time complexity. Please note that if the `Graph` were only to store the data in the `Flow::From` form, then calculating the `Flow::Into` form for this function would be an $O(n^2)$ operation; and vice versa. This is because all nodes and their respective adjacency lists would have to be checked for instances where the adjacency list contains the `node`, and thus leads to $O(|N|^2)$ for a dense graph. By storing both forms of the graph's data, this function instead maintains an $O(1)$ time complexity.

```
Data GetData(const Flow& flow) const;
```

The `GetData` method merely accesses the respective data field and returns the `Data` associated with said field. For example `GetData(Flow::From)` would return `from` and `GetData(Flow::Into)` would return `into`. Since these types of access operations have an $O(1)$ time complexity, this function has an $O(1)$ time complexity. As mentioned in the `GetList(...)` analysis, if the `Graph` were to only store the data in one form or the other, generating the form not stored would be an $O(n^2)$ operation. Again, by having data fields for each form, this function instead maintains an $O(1)$ time complexity.

```
Page GetPage() const;
```

The `GetPage` method returns the default page-rank of the graph. This is simply where each webpage obtains the same rank of $\frac{1}{|N|}$, where $|N|$ is the number of nodes in the graph. Because the `Page` must be constructed manually, the function must iterate through all n nodes in the graph. As a result, the time complexity of this function is $O(n)$. Please note that while the function does utilize the `GetRank()` method, said method is $O(1)$ and thus has no effect on the overall time complexity.

```
void GetPage(unsigned int power, Page& page) const;
```

The `GetPage` method recursively calculates the `Page`, based off of the *Page-Rank* algorithm. The method must iterate through all n nodes in the graph and calculate the ranks for each n number of adjacent nodes (of the `Flow::Into` form). This results in an $O(n^2)$ operation. Additionally, because this function recursively calls the function p number of times, where p is the `power` provided, the aforementioned operation is called p times. Therefore, this method has an $O(p * n^2)$ time complexity. As mentioned in the `GetList(...)` analysis, if the Graph were to only store the data in one form or the other, generating the form not stored would be an $O(n^2)$ operation and thus leads to $O(p * n^3)$ for this function. Again, by having data fields for each form, this function instead maintains an $O(p * n^2)$ time complexity.

Reflection

The most significant issue for this project was my inability to convey the problem properly before beginning code implementations. I spent a rather great portion of my time on this project going through unsuccessful code iterations because I didn't fully understand the problem, and thus I didn't fully understand what functions or data fields were truly necessary. After wiping the slate clean, researching, and re-designing my code from the ground up, I found that a lot of the functionalities I had previously built were completely unnecessary. I have learned that in the future I should spend a lot more time delving into the problem before trying to design poorly implemented solutions.

Similarly, and this is despite my research, I still feel there is significant time complexity optimization potential. I have read that the true *Page-Rank* algorithm essentially has an $O(n)$ time complexity. I'm uncertain as to whether this is because their implementation uses an adjacency matrix instead (which I would have preferred), but even after finishing the project I still think there are optimizations that could be made. For example, while I do believe I made good justifications for why I choose the data structures I did (i.e., `std::map`, `std::vector`), I also think there might be potential optimizations in choosing a different set of data structures.

Finally, I would try to implement a solution that maintains the lower time complexities achieved while also storing only one version of the adjacency list. The two-data-field implementation makes the rest of the implementations far easier to read and design, but it also is not a practical solution for sufficiently large data sets. This would almost certainly be my main focus if I had to do this project again.