Yosef Gilbert

Time Complexity Analysis

In the following analysis, let p denote the power iteration of the algorithm, where $p = 1$ describes the base case where no matrix multiplication takes place. Let n denote the amount of lines inputted into the program, equivalently, the amount of edges of the graph.

Since the graph is given to be connected, the number of vertices cannot exceed $n + 1$, so asymptotic analysis using the amount of vertices as the measurement of graph size is equivalent to using the amount of edges as the graph size. Even if the graph is not connected, the amount of unique vertices will certainly be bounded by $2n$. Assume that string lengths are constant, because URLs have limited length.

insert

$O(log(n))$
This function attempts to find the "from" vertex in the map representation of the graph. In the worst case, "from" does not exist in the map so the search goes through the entire height of the underlying red-black tree data structure, taking $O(log(currentSize))$ time. In the worst case, the last insertion into the map is being performed so there may be n vertices already in the map. After, constant time operations occur to insert the new node and to push_back the vector.

pageRank

$O(pn^3)$
First the algorithm finds the base case ranks r(1), which takes $O(n)$ time. Then it applies matrix multiplication p times, using the new ranks generated each time. Each multiplication has two nested for loops, each running through the size of the graph in vertices. Computing the value is a constant time operation, but using find() to check if a node is in a vector takes linear time with respect to the vector size. In the worst case, this vector has $n - 1$ nodes.

printRanks

$O(n)$

This function runs in linear time with respect to the number of vertices, because it must visit each one to print its rank. Since the graph is given to be connected, the number of vertices cannot exceed $n + 1$. This worst case represents the sparsest graph that can be built with n edges.

main

$O(pn^3)$
First some constant time operations are executed. After creating an AdjacencyList object, insert is called for each edge. The time complexity of insert depends on how many vertices are already in the map. If the vertices are inserted in a way such that after any amount of insertions the graph is always connected, then at most one unique vertex is introduced with each insertion, so the time complexity will be
$log(2) + log(3) + log(4) + ... + log(n) = log(n!) < nlog(n)$
If insertion is not subject to this constraint, it will still be $O(log(n!))$.
Next, main calls pageRank() and printRanks(). Since pageRank() has the highest order of growth, the overall time complexity is the same as that of pageRank().

Reflection

The data structure I used to implement the graph is an adjacency list. It makes sense to use an adjacency list to optimize memory usage when representing a sparse graph. Groups of websites will likely form a sparse graph because it is not usually the case that each website links to many more in the group. Specifically, I used a map with strings as the keys and vectors of strings as the values. A map is useful because the output was required to be sorted, and the map uses a tree to keep data sorted without making inserting too expensive. To keep track of the rankings, I used a separate vector of doubles. I wanted this separate so the pageRank function could iterate its algorithm over the rankings without having to change the map each time.

From this assignment I learned how to implement a graph, which will be useful for many projects I'm interested in. I learned what the advantages of different graph implementations are, and saw that an adjacency list is usually ideal because it strikes a balance between ease of implementation and having good space and time complexity. I also had to familiarize myself with the definition of matrix multiplication in order to implement the algorithm, because I have not taken linear algebra yet.

I'm fairly happy with how I completed the project, but If I were to start over I might try to research Strassen's algorithm to improve the efficiency of the matrix multiplication. I would also add functionality to return a list of the correctly rounded ranks with the corresponding website for easier testing, instead of printing them out directly.