Page Rank Documentation

By Steven Miller

Department of Computer Science and Engineering

This document lists all functions that are contained within the page rank algorithm

Note that helper functions associated with the following functions are not present in this document. Assume that they are O(1) for best, average, and worst case.

The algorithm works by the user inserting two "pages" for every input along with. These pages are strings that are the names of the websites they want to create edges between. Once the pages are inserted, the algorithm then creates two `node*` objects that are dependent on the two pages inserted respectively. These "nodes" are then inserted into a `<string,node*>` based map. Where the `string` is the name of the page inserted. Every node contains a two `std::vector<node*>` named "out" and "in". Once the node for the first and second websites are created, the second websites node is added to the "out" vector of the first node. The first websites node is then added to the "in" vector of the second node. Thereby creating an "edge" between them. This creates a graph of website nodes that are interconnected with each other and contain their own respective page ranks.

The reason this design was chosen is for two reasons:
1.It allows us to search for websites and directly grab their pagerank by their name
2.It is the simplest and least resource intensive way to implement this algorithm without using a matrix-based approach. Which wastes a lot of space for sparse graphs.

What I learned from this assignment is that not all problems are as hard as they seem. After the previous project (project 1), I got an early start on this project. As I was scared that it would take an excessively long time. However, while project 1 took me a total of two weeks to complete. This project took me a total of two days to complete. I seriously underestimated the workload of this project.

I would not do anything over again. This project went off without a hitch really. The implementation is satisfactory, and it works very well. Overall, I'm happy that this worked out well.

```
void adj_list::add_node(std::string from, std::string to)
```

This function is responsible for adding nodes to the adjacency list. This function has a best-case time complexity of O(1), and an average and worst-case time complexity of O(|V|). As even though creating and linking nodes is O(1), it must check to see if the "from" and "to" nodes already exist. In the best case, these nodes are at the front of the list. In the worst case, they're all the way at the end.

```
adj_list::node* adj_list::find_node(std::string name)
```

This function is responsible for finding a node in the adjacency list that matches the specified name. This function has a best-case time complexity of O(1), and an average and worst-case time complexity of O(|V|). In the best case, the node is at the front of the list. In the average case, it's in the middle (which, how far the "middle" is in the adjacency list is unspecified), in the worst case, it's all the way at the end.

```
void adj_list::pagerank(int iterations)
```

This function is responsible for calculating the pagerank for every node. This function has a best-case time complexity of O(1), and an average and worst-case time complexity of $O(|V|^2)$. In the best case, it only needs to perform a power iteration on one node. In the worst and average case however, it needs to perform a power iteration for every node in the adjacency list.

```
bool adj_list::check_if_edge(std::string from, std::string to)
```

This function is responsible checking if the nodes matching the specified names are linked together with at least one edge. This function has a best-case time complexity of O(1), and an average and worst-case time complexity of O(|V|). As it relies on the "find_node" function, which we discussed earlier has a O(|V|), and it also needs to check the "from" node to see if any of its "to" nodes match the specified "to".

```
int adj_list::get_index(node* n)
```

This function is responsible for getting the index of the specified node. This function has a best-case time complexity of O(1), and an average and worst-case time complexity of O(|V|). As in the best case, the node is at the front of the adjacency list. But in the average and worst case, it is at the middle or end.

```
adj_list::~adj_list()
```

This function is the destructor. It has a best, average, and worst case of O(|V|). As it will always iterate through the entire adjacency list to free memory.

```
int main()
```

This function is our main. It has a best case of O(1), when only 1 power iteration is needed, and an average and worst case of O(|V|), when an unspecified amount is needed.