

PageRank PseudoCode

```
// initializing page rank for further iterations

for (auto it : webPages)
{
    it->pageRank = (1 / (float)webPages.size());
}

//iterations generator of pageRank for convergence

void webGraph::pageRankIteration()
{
    for (auto it : inv_adjList)
    {
        it.first->tempPageRank = 0;
        for (auto node : inv_adjList[it.first])
        {
            it.first->tempPageRank += node->pageRank / node->hyperLinksCount;
        }
    }

    for (auto it2 : webPages)
    {
        it2->pageRank = it2->tempPageRank;
    }
}
```

```
//run x iterations, in this example 10 iterations for better convergence
```

```
//sum of all pageRanks will always = 1 (due to normalization)
```

```
int iterations = 10;
```

```
    for (int i = 0; i < iterations; i++)
```

```
    {
```

```
        pageRankIteration();
```

```
    }
```

```
//function to calculate CTR
```

```
void webGraph::CTRCalc()
```

```
{
```

```
    for (auto page : webPages)
```

```
    {
```

```
        if (page->impressions != 0)
```

```
            page->CTR = (100 * (page->clicks / page->impressions));
```

```
        else
```

```
            page->CTR = 0;
```

```
    }
```

```
}
```

```
//Function to calculate Score using pageRank and CTR
```

```
void webGraph::scoreCalc()
```

```
{
```

```
    for (auto page : webPages)
```

```
    {
```

```

        page->score = 0.4 * page->pageRank + ((1 - ((0.1 * page->impressions) / (1 + 0.1 *
page->impressions))) * page->pageRank + ((0.1 * page->impressions) / (1 + 0.1 *
page->impressions) * page->CTR) * 0.6));
    }
}

//assist tool used for sorting the search results for ordered Indexing
bool mainMenu::cmpScore(webPage *x, webPage *y)
{
    return x->score < y->score;
}

sort(relevantWebPages.begin(), relevantWebPages.end(), cmpScore);

//can be displayed after any search query in order of index(score)

```

Time and Space Complexity

(Ranking and Indexing)

- **Time complexity:**

1. Calculating the PageRank of each vertex.
 - $O(|V| * |E|)$ time, because it requires iterating over all the vertices and edges in the graph.
2. Updating the PageRank of each vertex.
 - $O(|V|)$ time, because it requires iterating over all of the vertices.

- **Space complexity:**

1. The vector of vertices
 - $O(|V|)$ space to store all nodes
2. The graph represented as an adjacency list
 - $O(|V| * |E|)$ space to store all nodes and their edges.

Data Structure Usage

The primary data structures employed in my Search Engine were maps and vectors due to their distinct advantages and features. These data structures were chosen to optimize time and space complexity while improving code organization and simplicity. Maps were particularly beneficial for their efficient key-value lookup mechanism. They proved valuable in associating URLs, website names, or keywords with relevant information, such as ranking, scores, or internal variables. By utilizing maps, I could quickly retrieve information, making it ideal for tasks such as indexing and sorting search results. I employed pointers to structs of webPages that contained all necessary internal variables. Combining the built-in map sorting operation with a compare score function enabled automatic updating of search results based on relative scores, simplifying the process of indexing the results for output. On the other hand, vectors served as dynamic arrays with fast random access and efficient insertion/deletion times. These properties made them well-suited for tracking multiple websites efficiently. Vectors also offered easy sorting, which facilitated the storage of keywords, scores, and other variables in a compact and convenient manner. By leveraging the strengths of both data structures, I achieved optimized search and operational efficiency when handling the stored data. This approach allowed for a coherent flow of operations, enhancing the functionality and usability of my search engine without requiring significant alterations to the core structure.

Design Tradeoffs Justified

In the process of designing my search engine, I made a couple of tradeoffs that had minor impacts on certain functionalities in the design. For one, I limited the maximum word search to two words. This decision was made based on a balance between search accuracy and computational efficiency. By restricting the search to a maximum of two words, I could simplify the search algorithm and reduce the complexity of matching and ranking results. While this limitation may occasionally result in less precise search outcomes for more complex queries, it significantly improved the speed and responsiveness of my search engine.

Another tradeoff was related to the website impression count, which would increment by two for each interaction. When a user selects a website from the list, it behaves as if the person had opened the link, and upon returning to the list, the impression count increments again. Although this approach may slightly overstate the impression count, it aligns with the user's behavior of entering a website, then navigating back to explore other options. Additionally, users have the freedom to click on a website multiple times, which would accurately increment the click counter for that specific website regardless of the impressions, making it practically negligible overall.

Altogether, these design tradeoffs were justified by considerations of computational efficiency, search speed, and aligning with user behavior. By making these decisions, the search engine could deliver reasonably accurate results within acceptable performance limits, while still providing a user experience that resembled real-world browsing patterns.