Mario Ghaly

5/5/2022

Search Engine Project Report

1. Since the nodes are URLs, so the graph implementation had to be different from the implementation of a graph with nodes of integers. I used the idea of mapping. I created struct Node with many data

Struct Node{

//store the impressions, clicks, Rank, CTR, score, URL, keywords, index in the vertices vector in the graph, and vector<int> nodes to store the nodes going to this node, so when needed to calculate the Page Rank, the algorithm will go directly to the source nodes without checking any redundant nodes -- in case of using the adjacency list or matrix

//default constructor

Node(string str, int ind){} //for initializing the nodes with the URLs and indeces

};

Class Graph{

private:

Int V //number of nodes

vector<vector<int>> adjlist;  //adjacency list stores the indices of nodes

     vector<int> map;                   //a vector for sorting the indices of the nodes according to their score//

Public:

void addEdge(string src, string dst) { //the function takes the two links

              int start = -1, end = -1;

        for (int i = 0; i < vertices.size(); ++i) {

            if (vertices[i].link == src)     //checks if this link exists or not

                start = i;

            if (vertices[i].link == dst)

                end = i;

        }

        if (start == -1) {      //if it doesn’t exist, create a new node and its index is equal to the vector size//

            start = vertices.size();

            vertices.push\_back(Node(src, start));

            map.push\_back(start);   //stores the indices

        }

        if (end == -1) {     //same

            end = vertices.size();

            vertices.push\_back(Node(dst, end));

            map.push\_back(end);

        }

        adjlist.resize(vertices.size());   //adjust the size of the adjacency list by the new size of vertices vector//

        adjlist[start].push\_back(end);  //put the index in the adjacency list

        vertices[end].nodes.push\_back(start);  //put the source link in the nodes vector of the destination link, for the page rank

        V = vertices.size();  //update number of nodes

    }

int Find\_index(string link) {   //to search the vector of vertices in the graph by link to get the index//

        for (int j = 0; j < V; ++j)

            if (vertices[j].link == link)

                return j;

    }

//For calculating the page rank

for (int j = 0; j < 7; ++j) {

            for (int i = 0; i < V; ++i) {

                double sum = 0;

                for (int r = 0; r < vertices[i].nodes.size(); ++r) {  //this loops over the src nodes

               sum += (vertices[vertices[i].nodes[r]].Rank / adjlist[vertices[i].nodes[r]].size());

  //Here I used the nodes vector for a better time complexity, and the size of the row of the source node.

                arr[i] = sum;  //this array is for storing the values of the current iteration

            }

//Here I loop once again to update the page ranks of vertices

}

};

1. Time complexity of adding each edge is V in the worst case, so creating the graph will be of time complexity O(V\*E).

Time complexity for the ranking algorithm in the worst case will be O(7 \* N \*N) = )(N\*2)

1. My main data structures are as follows:
   1. Node: structure containing 2 integers, 4 doubles, 1 vector<string> for the keywords, and 1 vector<int> for storing the indices of nodes going to that node as illustrated above
   2. Graph: class containing 1 vector<Node> for storing the data of the nodes, adjacency list represented as a vector<vector<int>>, and a map vector<int> for mapping the URLs with the indices
   3. 1 vector<string> for storing the searched keywords, and vector<int> for the indices of the URLs
2. 1 tradeoff I made gave me a great advantage in space complexity and in making the code simpler and neater. In the beginning, I made a vector<Node> called ranked to store the vertices according to their score so that it will be easier to loop over websites when searching for a keyword. In that way, I had both vertices by their normal indices, and in their order according to the score, and I can use both. However, there were 2 problems. The first is that another vector of Nodes takes a huge amount of space, and it was a waste of space because there are more efficient ways. The second problem is that in order to update anything in the normal vertices vector of Node, the ranked vector should be updated as well, so it was not proper coding to keep repeating the same line of code. The tradeoff was to replace the ranked vector with a mapping vector<int> that still ranks the nodes according to their score, but it puts only the index. It is the same idea I applied with the adjacency list.

The other tradeoff is that there was an idea of removing the original vector of nodes and keeping the ranked one. However, this will result in time complexity O(N) if we wanted to add another node in the graph (in my implementation). Thus, I preferred to have a better time complexity.