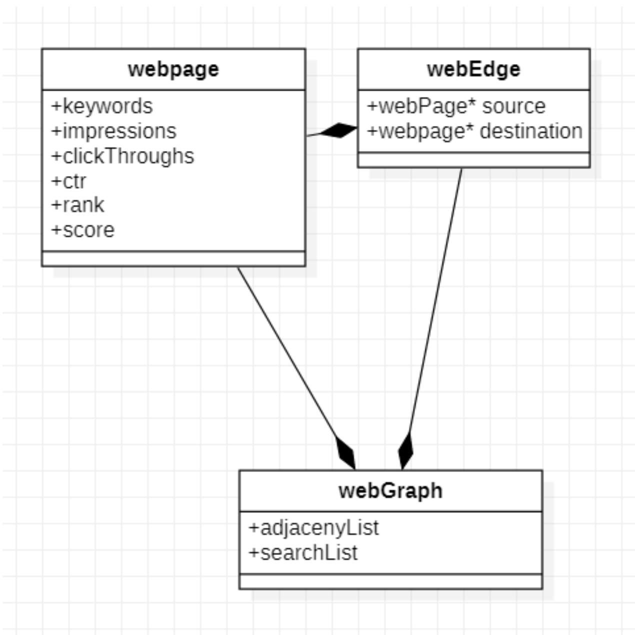
Ahmed Waseem Raslan 900211250

Simple Search Engine Report

Rudimentary class diagram of relevant classes and attributes;



Webpage: class containing data relating to webpage

Keywords (vector<string>): contains Keywords of class.

Impressions (integer): impressions of a Webpage.

clickThroughs (integer): click throughs resulting from user activity

ctr (float): calculated click through rate

rank (float): normalized page rank

score (float): calculated page score

webEdge: class to construct web Graphs and to ease parsing edge.csv files

source (webpage): Edge source of type webpage pointer.*

destination (webpage): Edge Destination of type webpage pointer.*

webGraph: graph class containing the web of webpages, represented using an adjacency list

adjacenyList (map<webpage, list<webpage*>>): a map of webpages and the list of webpages its pointing to*

searchList (map<string, set<webpage>>): map of keywords and the set of webpages that contain that keyword. Initialized at first then used to cut down complexity of search.*

Constraints:

Webgraph is not dynamic.
Search is based only on keywords the webpages contain.

INTIALIZATION:

parsing of web graph from CSV files has a time complexity of $O(N^2)$ and a space complexity of $O(N(K+E))$ K *being the number of keywords* E *being number of edges*.

Since web graph consists of a map were webpages are keys and a list of webpages are their elements it has a space complexity of $O(NE)$ E *being number of Edges*. Time for constructing web graph is $O(NLogN)$.

Initialization calls functions that calculates ctr, ranks, scores.

PAGERANK :

Algorithm used is based on the following sources:

[1] Victor Lavrenko, L. V. (n.d.). Web search 7: sink nodes in PageRank.
YouTube. https://www.youtube.com/watch?v=_Wc9OkMKS3g

[2] Global Software Support. (n.d.). PageRank Algorithm - Example.
YouTube. https://www.youtube.com/watch?v=P8Kt6Abq_rM

[3] Global Software Support. (n.d.). PageRank Algorithm - Final Formula.
YouTube. https://www.youtube.com/watch?v=EvVPoDMDTM8&list=PLH7W8KdUX6P2n4XwDiKsEU6sBhQj5cAqa&index=8

Additional features:

Sink node implementation based on algorithm described by Viktor Lavrenko on YouTube[1]
Damping factor of 0.15 (probability of a user to leave the current webpage).

Calculating one iteration of page rank has time complexity of $O(NT)$ T *being number of keys in the transpose map*. And calculating all iterations is bounded by $O(N^2T)$ However the true complexity in practice is lower since algorithm stops iterations if it finds no change is happening to the page rank.

Remarks:

Using a big Graph of 50 nodes and 16 sink nodes, there is still a page rank bleed after implementation of damping and sink node handling. I suspect it is a an error in implementation or algorithm. However without sink node handling the algorithm works as intended.

SEARCH :

As mentioned, a search List is initialized at the beginning of the program
Time complexity of initialization: $O(NM^2)$ N being number of nodes, M being number of keywords in graph (M squared since the worst case is that keywords are repeated M times, Meaning all webpages have the same number of keywords).

The result is a map that can be accessed in $O(Log_2 N)$ time to search for a single keyword.

For multiple keywords in search query the complexity is $O(\text{QLog}_2 N)$ where Q is number of keywords in search query.

Search is done by normalizing the given query into a vector of strings. This allows to iterate on keywords and checking for the tokens easily to be able to process large queries.

PSEUDOCODE:

Ranking:

```
Map<page, float> calculate_ranks_iteration(transpose, sinkNodes)
{
    Map<page, float> rankstemp;
    float sumOfSinkNodesPR = 0;
    float sumOfPR = 0;
    float s = 1;

    for (i -> sinkNodes)
    {
        sumOfSinkNodesPR += sinkNodes[i].rank;
    }

    s = (sumOfSinkNodesPR / _adjList.size())*DAMPING_FACTOR;

    for (I -> N)
    {
        //for every webpage in the map its PR is the sum of the Ingoing Nodes' PRs
(outgoing since we are looping on transpose) over their number stored in temp map

        rankstemp[i] += s;
        rankstemp[i] *= DAMPING_FACTOR;
    }
    return rankstemp;
}


void calculate_ranks()
{
   //get transpose of graph

    map<webpage*, float> rankstemp;
    map<webpage*, float> rankstemp2;


    //PR intialization
    for (I -> N)
    {
        Node[I].rank = (1.0 - DAMPING_FACTOR) /  graph.size();
    }

    //get sink nodes

    //iterations to size - 2 ( -2 because iteration 0 is already done )
```

```
    for (I -> N -1)
    {
        rankstemp = calculate_ranks_iteration(transpose, RanksSinkNodes);

        if (rankstemp == rankstemp2)
        {
            break;
        }
        else
        {
            //ranks updated with calculated ranks
            for (I -> N)
            {
                Nodes[I].rank = rankstemp[I];
            }
            rankstemp2 = calculate_ranks_iteration(transpose, RanksSinkNodes);
            I++;
        }
    }

    //normalizing pageranks
    norm = sort(rankstemp);
    int temp = 1;
    for (I -> norm.size)
    {
        if (current node in rankstemp has same rank value as the next node)
        {
            currentNode->_rank = temp;
            nextNode->_rank = temp;
            i++;
        }
        else
        {
            currentNode->_rank = temp;
        }

        temp++;
    }

}
```

For indexing pages to user:

```
vector<webpage*> results = graph.search(normalizeQuery(userinput));

cout << "\nYour Results\n";
int index = 1;
for (I -> results.size())
{
       indexedResutls[index] = results[i];
       results[i]._impressions++;
       results[i].update_score();
       cout << index << ". " << webpage->_url << " score: " << results[i]._score << "\n";
       index++;
}
```