Name : Kirollos Ehab Zikry

ID : 900213037

Project report

* + 1. **The pseudo-code for your indexing and ranking algorithms**

# Reading Files Algorithm

getGraph():

Open “webGraph.csv” file

For each line in the file:

Split the cells inside the line by a comma to get the start and adjacency list

Add start as a key to the graph map and adjacency list as its value

Add start to incoming\_links map and update the incoming links for each node in the list

(Do the Same for “keyword.csv” inside getKeywords(), “impression.csv” inside getImpressions(), and “CTR.csv” inside getCTR())

# Ranking Algorithm

page\_rank():

size = graph.size()

For each entry in graph:

pagerank[entry.first] = 1.0 / size

For j = 0 to 1:

For each entry in pagerank:

temp\_rank = 0.0

For each link in incoming\_links[entry.first]:

temp\_rank += pagerank[link] / graph[link].size()

entry.second = temp\_rank

get\_websites(query):

final\_result = empty vector of pairs (string, double)

result = search\_table[query]

FOR each i in result:

score = (0.4 \* pagerank[i]) + (((1.0 - ((0.1 \* impressions[i]) / (1.0 + (0.1 \* impressions[i])))) \* pagerank[i]) + ((1.0 - ((0.1 \* impressions[i]) / (1.0 + (0.1 \* impressions[i])))) \* CTR[i])) \* 0.6

final\_result.push\_back(pair(i, score))

Sort final\_result based on score in descending order

RETURN final\_result

print\_results(result):

i = 1

FOR each entry in result:

PRINT i + "- " + entry.first

impressions[entry.first]++

i++

clicked = 0

clicked2 = 0

READ clicked

IF clicked = 0:

RETURN true

ELSE IF clicked = -1:

RETURN false

ELSE:

PRINT the result

CTR[result[clicked-1].first]++

READ clicked2

SWITCH clicked2:

CASE 1:

RETURN print\_results(result)

CASE 2:

RETURN true

CASE 3:

RETURN false

update\_registry():

Open "CTR.csv" file for writing

For each entry in the CTR map:

Write the website and its CTR value to the file

Open "impression.csv" file for writing

For each entry in the impressions map:

Write the website and its impression count to the file

# Process the user's search query

process\_query(query):

flag = True

if query starts with double quotes:

new\_query = extract the query inside the double quotes

result = get\_websites(new\_query)

flag = print\_results(result)

else if query contains "OR":

Split the query by "OR" to get two separate queries

result1 = get\_websites(first query)

result2 = get\_websites(second query)

result = combine result1 and result2 (remove duplicates)

flag = print\_results(result)

else if query contains "AND":

Split the query by "AND" to get two separate queries

result1 = get\_websites(first query)

result2 = get\_websites(second query)

result = find common websites in result1 and result2

flag = print\_results(result)

else:

Split the query by space to get two separate queries

result1 = get\_websites(first query)

result2 = get\_websites(second query)

result = combine result1 and result2 (remove duplicates)

flag = print\_results(result)

return flag

* + 1. **A time and space complexity analysis for your indexing and ranking algorithms**

1. getGraph():
   1. Time Complexity: O(N)
   2. Space Complexity: O(N)
2. getKeywords():
   1. Time Complexity: O(N)
   2. Space Complexity: O(N)
3. getImpressions():
   1. Time Complexity: O(N)
   2. Space Complexity: O(N)
4. getCTR():
   1. Time Complexity: O(N)
   2. Space Complexity: O(N)
5. page\_rank():
   1. Time Complexity: O(N^2)
   2. Space Complexity: O(N)
6. get\_websites(string query):
   1. Time Complexity: O(N)
   2. Space Complexity: O(N)
7. print\_results(vector<pair<string, double>> result):
   1. Time Complexity: O(N)
   2. Space Complexity: O(1)
8. update\_registry():
   1. Time Complexity: O(N)
   2. Space Complexity: O(1)
9. process\_query(string query):
   1. Time Complexity: O(N)
   2. Space Complexity: O(N)

* Time Complexity: O(N^2)
* Space Complexity: O(N)
  + 1. **The main data structures used by your algorithm**

Most of the data structures I used are unordered\_maps. This structure provides fast access to key-value pairs using hashing, which makes it a good choice for storing and accessing large amounts of data.

* + 1. **Any design tradeoffs you made along with their justifications**
* The code utilizes unordered maps for efficient data retrieval and modification, sacrificing the order of elements.
* Error handling is limited or absent, prioritizing code simplicity over handling unexpected scenarios.
* The code lacks modularization and a clear code structure, likely for simplicity in the given context.