Ali Ashraf

CSCE220301 - Anlys and Design of Algorithms Lab (2021 Fall)

Dr Mohamed Alhalaby

Sunday, 21 November 2021

Search Engine Report

Table of Contents

Data Structures and Design TradeOffs

The following is a description of the major data structures that were used throughout the implementation of the search engine.

Acknowledgement: while the implementations of classes as data containers might be memory-inefficient for high scale operations, this effect rests negligible given the small-scale nature of such an assignment.

WebPage class

Was constructed to define and associate attributes and, inherently, all the accommodating operations such including updating and retrieving from a centralized entity - that is, the WebPage object. Besides that, implementing the Object-Oriented Programming allowed for easy debugging and easy maintainability of the code. Contained member functions, mainly setters and getters, that facilitate the updating and retrieval process of a page's attributes. The setters execute in the main program, when reading from a file, to set or update a WebPage's URL, impressions, clicks, and PageRank. Getters are essential for retrieving defining attributes such as URL and vertex number (which is important for indexing WebPages in the Graph class). It is worth noting that reading and saving of CTR to a file was unnecessary since the function to calculate CTR executes automatically whenever a click, impression, or PageRank gets updated.

Unordered Map of WebPages

Storing the WebPage objects demands a data structure that allows for a nearly constant access time. Hence, an unordered map was the ideal choice seeing as they are associative containers – that is, they store key-mapped values. Contextually, URLs were used as keys to access the corresponding object and, since the sets of URLs and WebPages are bijective, there

exists one and only one distinct URL for each WebPage object. Thus, no collisions occur. Accordingly accessing, manipulating, or retrieving data from said WebPages can be done conveniently by their names and in an average time complexity of ***O(1)***. This was especially useful since the data – be it impressions, keywords, or PageRank – are exclusively indexed by website name in the CSV files; resultantly, reading impressions, clicks, keywords, and PageRank was an easy task since no extra mapping between names and objects were required.

Graph class

Was also defined to minimize code deficiencies, including code duplications. The Graph class receives a set of edges – comprised of source and destination nodes and populates an adjacency list - technically a vector of vectors - accordingly such that PageRank can function suitably.

Adjacency List

An adjacency list was implemented to represent the connections between each of the websites, the latter of which received as WebPage objects. Adjacency list were represented as vector of vectors, where the source is the first object in the list, and any subsequent nodes represent an outgoing edge from the source to them. Despite its challenging implementation, adjacency list was the optimal solution, by contrast to adjacency matrix, seeing as the latter's time and space complexity stand at a constant time of $\Theta(n^2)$ whereas the former's space complexity is more efficient depending on the length of each individual vector which is, disregarding worst-case complexity (a complete graph), O(V + E) whereby each vertex or source is visited only once plus visiting any subsequent destination node or iterating over an edge. Also, the same complexity is true for storing the adjacency list.

Trie class

The Trie class was constructed, likewise, to minimize code deficiencies. The Trie data structure was chosen as a storage and retrieval medium of keywords, which is arguably the most efficient and apt data structure for said function. Specifically, they are used in the implementation of text-correcting software and dictionaries. Trie is a digital tree the stores strings as character-nodes where every node represents a character whose children are the possible character combinations (the maximum size of which is the alphabet size). Therefore, insertion and search operations depend strictly on the length, ***L***, of the search query, resulting in a time complexity of ***O(L)***. Each node was stored as a struct of:

$$\{pointers\ to\ characters, vector < string > pages, isLeaf\}$$

Bool isLeaf indicates whether the current node is a leaf node, which corresponds to a complete string or word. If so, the search function returns the corresponding webpages (contained in the vector<string>).

Positive implications

- Insertion and search are done in a constant time of Θ(L) which is rather fast and more efficient than mere arrays and the likes of self-balancing trees such as binary-search Trees and AVLs.
- By contrast to self-balancing trees, it does not require hashing and, consequently, evades the tiresome, time-consuming collision-handling process; therefore, it is faster.

Drawbacks

- Seeing as every node in Tries contains a huge number of other node pointers – ranging from 26 to 255, depending on the size of the alphabet – one can conclude that Tries are

not memory-efficient and consume a lot of unnecessary space as a string-storing data structure.

Pseudocode

PageRank Algorithm

The following implementation is majorly the product of my thinking, testing, and debugging, with the very exception of the logic behind zeroing the current PageRank for subsequent iterations of PageRank and the choice of the damping factor.

```
void PageRank( ):
    for i = 0 → n-1:
        visitedAsNode_i = false;
        currentPr_i = 0;
    for i = 0 → n-1:
        string root = adjList_{i,0};
        int rootIndex = map[root];
        double c = adjList_{i | size} − 1;
        for x = adjList_{i | begin} → adjList_{i |end} :
            If (*x  != root):
                int xIndex = map_x;
                If (visitedAsNode_xIndex == false):
                    visitedAsNode_xIndex = true;
                    currPr_xIndex += ( 0.85 * ( oldPr_rootIndex / c ) ) + ( 0.15 / n );
                Else:
                    currPr_xIndex += ( 0.85 * ( oldPr_rootIndex / c ) );
    copyNewPrToOldPr();
    Ofstream myFile.open("pagerank.csv");
    unordered_map < string, int >::iterator = map_begin;
    for i = 0 → n-1:
        myFile << iterator → first<< "," << currPr_i<<endl;
        iterator ++;
    myFile.close( );
```

Analysis

Brief explanation about PageRank

- The choice of the damping factor:
    - The damping factor was chosen to be 0.85 as means of correspondence with the standards set by Google, it assumes that 80% of the time, a random web surfer will follow hyperlinks while the remaining 20% will be spent terminating the current sessions completely and instead navigating to new web pages randomly.

Time and Space Complexity

The algorithm is comprised of five loops. The first one is responsible for preprocessing, whereby it zeroes the new PageRank of all webpages before the commencement of the current iteration; this runs in a linear time of ***O(n)***, obviously. The fourth loop copies the current iteration's final PageRanks to an oldPr array before getting zeroed in the following iteration. This also runs in a time complexity of ***O(n)***. The fifth loop writes the new PageRanks to a file, "pagerank.csv" in a time complexity of ***O(n)***.

- The second (parent) loop traverses over source nodes and fixates each one whereas the third (child) loop, which is nested, goes over the pointed-to nodes and updates them with the corresponding normalized PageRank of the root.
    - The parent loop depends on the size of the source WebPages or, contextually, **v**ertices.
    - The child loop is contingent on the number of destination sources or, alternatively, the number of outgoing **e**dges from a certain vertex.
    - As a result, their combined time complexity depends on the cardinality of vertices by the cardinality of outgoing edges whereas the space complexity depends on the sum of vertices and edges.

∴ The time complexity of this algorithm is $\boldsymbol{O(3n + (|Vertices| * |Edges|))}$

$\boldsymbol{\therefore \Omega\ (|V| * |E|)}$

∴ The space complexity of this algorithm is $\boldsymbol{O(|V| + |E|)}$

However, given a full, complete graph in which every node is connected to every other node (the worst-case scenario), the complexity will rise to match that of an adjacency matrix:

$\boldsymbol{\therefore O\ (n^2)}$

Yet, such a state is impossible to realize seeing as such an unrealistic model assumes that all webpages link to every other webpage; therefore, one can confidently dismiss such a proposal and, otherwise, accept the fact that the average running time is of $\boldsymbol{O\ (|V| * |E|)}$**.** By contrast, an adjacency matrix requires a fixed traversal time of $O(n^2)$. For the sake of clarity, take the following web graph:

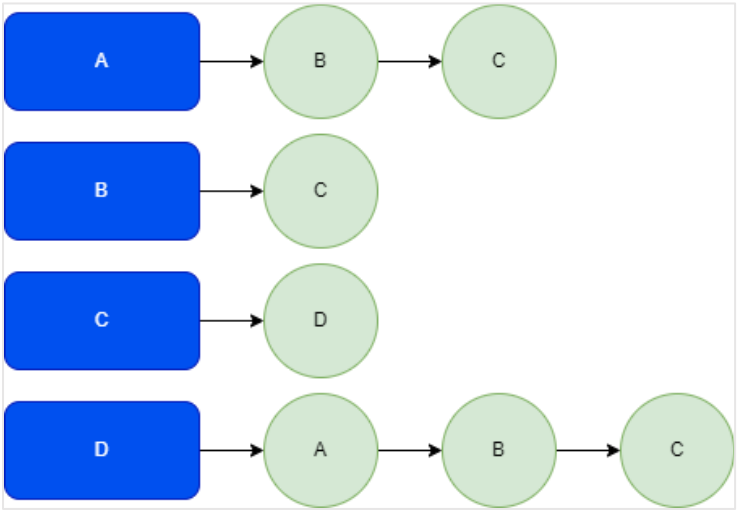

Figure 1 Diagrammatic representation of an adjacency list

Traversing through the above adjacency list, yields the following number of time complexity:

$$= |V| * |E|$$

$$= 1 * 2 + 1 * 1 + 1 * 1 + 1 * 3$$

$$= 1 * (2 + 1 + 1 + 3)$$

$$= 1 * 7 = 7$$

By sheer contrast, the traversal of such a graph via an adjacency matrix would necessitate the construction of a (V * V) two-dimensional array, thereby yielding a traversal time complexity of $O(V^2) = 4^2 = 16$. On another note, space complexity for the adjacency list, as denoted by $\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$, is (4 + 7) equaling 11 whereas that of the adjacency matrix equals 16. Evidently, the adjacency list implementation is over twice as efficient in terms of time and 35% more efficient in terms of space as an adjacency matrix. Since PageRank executes once during the startup of the program, one can simply disregard any comparisons as irrelevant. Nevertheless, since the primary concern is in regards with time complexity, one cannot simply dismiss the fact that adjacency list could be less than an adjacency matrix by one or half an order of magnitude.

Indexing Algorithm

The following code was obtained from techidelight.com. Core amendments were carried out to the code to enable the association of WebPage URLs and keywords, namely in the Node struct, insertion and search functions.

```
struct Node{
    bool isLeaf;
    Node* character[CHAR_SIZE];
    vector<string> pages;
};
void insert (string key, string page):
    Node* curr = &node;
    for i = 0 → key_length :
        if(curr → character[key_i] == nullptr):
            curr → character[key_i] = new Node();
        curr = curr → character[key_i];
    curr → isLeaf = true;
    curr → pages.pushBack(page);


vector<string> search(string key){
    vector<string> vec;
    vec.resize(0);
    If (this → node_character == nullptr):
        return vec;
    Node* curr = &node;
    for i = 0 → key_length :
        curr = curr_character[key_i]
        if (curr == nullptr):
            return vec;
    return curr_pages ;
```

Complexity Analysis

Time Complexity

One must analyze both the insertion and search algorithms when it comes to complexity analysis. Both must be put into perspective since they operate similarly.

Insertion

- Receives as input the keyword to insert and corresponding associated page name
- Inserts each character of the keyword as a character and marks the last character's node as leaf, associating the inserted page name via pushing back to the node's vector of pages.

∴ Each Insertion operation depends strictly on the length, ***L***, of the received keyword!

∴ Time complexity is ***O(L)***

Searching

- Receives as input a string query
- Keeps traversing the Trie, character by character
    - If the last character is reached and it is not a leaf node, it returns an empty vector, which means that no results were found
    - If the last character is reached and it is a leaf node, it returns this node's vector of pages

∴ Each search operation, likewise, depends strictly on the length, ***L***, of the query, which in the worst-case is the term "*pneumonoultramicroscopicsilicovolcanoconiosis*", the longest word in any of the major English language dictionaries, which is 45 characters long, according to Grammarly(https://www.grammarly.com/blog/14-of-the-longest-words-in-english/).

∴ Time complexity is ***O(L)***

Space Complexity

- A given query is ***L*** characters long, whereby for each character there is a node that is of the alphabet size.

∴ Therefore, the space required for storing one word is L* $Alphabet_{size}$

∴ The space complexity for storing N keywords is equal to

$$\boldsymbol{O(N * L * Alphabet_{size})}$$

Clearly, space is a tradeoff that the Trie data structure inherently brings to the table since for every node there exists pointers to an alphabet which can vary from just 26 characters to 128 and even 255 characters. However, the nearly constant access time of insertion and searching cause an efficient storage and retrieval of the search results which, altogether, justify the high memory consumption that inherently introduced by the Trie data structure.

Bibliography

There are no sources in the current document.