|  |
| --- |
| AUTOCOMPLETE SEARCH |
| MILESTONE 1 |
| **Prepared By: Avanish Kumar Singh (MT14004)** |

**TABLE OF CONTENTS**

1. Introduction
2. Theoretical Idea
3. TRIE
4. Radix Tree
5. Tools, Techniques and Technologies Used
6. Dataset
7. Evaluation Metrics
8. Assumptions / Special Cases
9. Queries to be used for evaluation (test cases)
10. Progress
11. References

**INTRODUCTION**

***Autocomplete is a feature in which a search engine (or any application) automatically predicts the remaining characters in a word or phrase based on what has been typed by the user.***

***E.g.*** If the user types *“sac”*, the system displays some suggestions/words such that *“sac”* is a prefix of all those suggested words, like *“sachin”*, *“sack”*, etc.

***Advantages of autocompletion:***

* Autocomplete speeds up human-computer interactions when it correctly predicts words being typed.
* The key technical insight is that people type slowly, but read quickly, typically taking 300 milliseconds between keystrokes, but only 30 milliseconds (a tenth of the time!) to glance at another part of the page. This means that the user can scan suggestions while he types.
* The most obvious change is that the user gets to the right content much faster than before because he doesn’t have to finish typing his full search term, or even press “search.”
* Another shift is that seeing suggestions as the user types helps him formulate a better search term by providing instant feedback. He can now adapt his search on-the-fly until the suggestions match exactly what he wants.

*The end-product of this project will be a browser-based web application that will have a simple user interface having a text-box. The user will type into the text-box, and on-the-fly the GUI will be updated with some words. These words will be suggestions based on what the user has already typed. The list of suggested words/terms will be updated after each key-press event.*

**THEORETICAL IDEA**

The basic idea behind the design of my autocomplete search system is the use of a prefix-tree data structure such as **TRIE** (derived from reTRIEval).

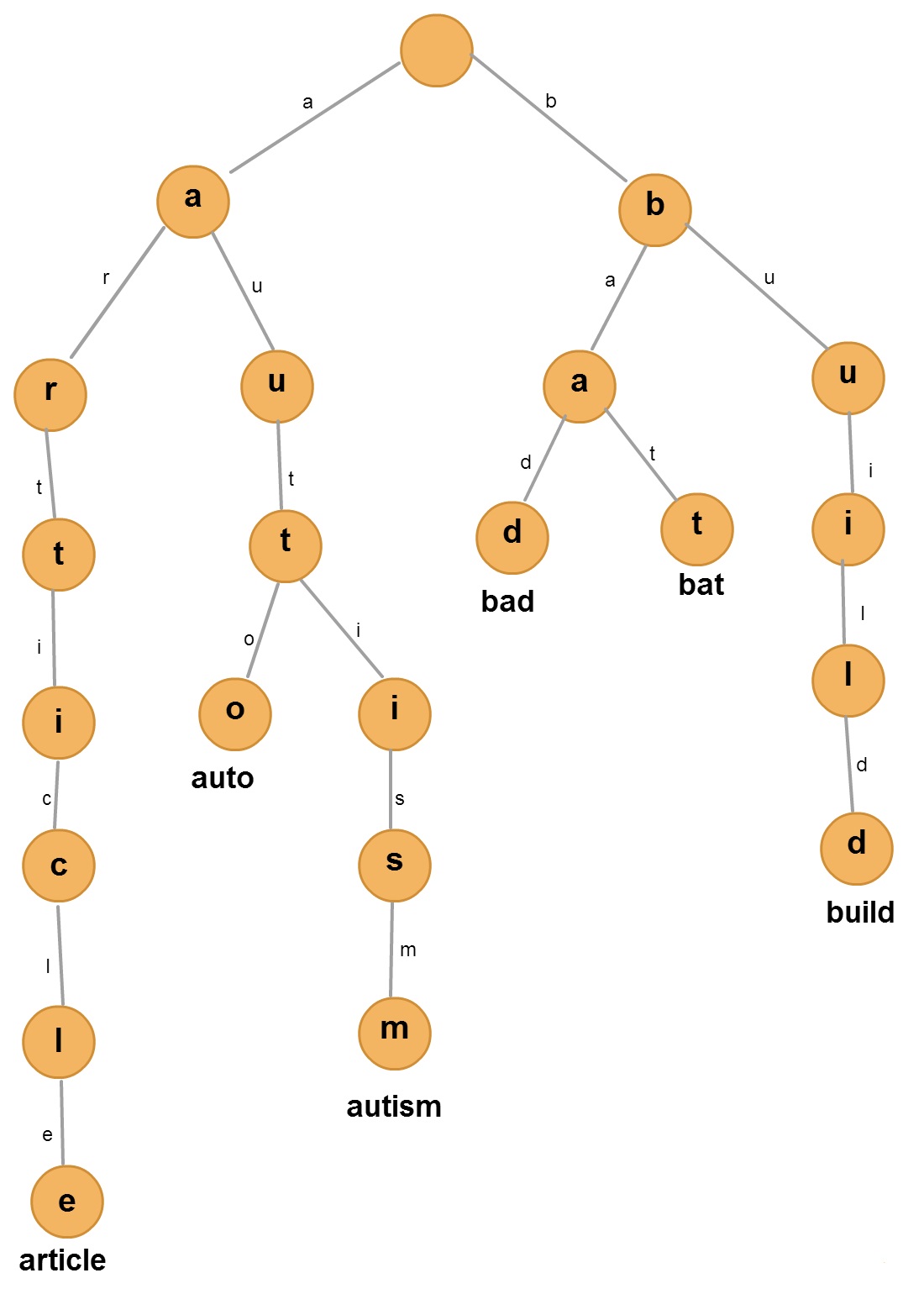
*For better space utilization, I will implement a modified version of Trie called the* ***Radix Tree****. The radix tree is basically a* ***compressed Trie****.*

***Simplified approach that I’ll use for solving the main problem (of producing autocomplete results to the user):***

* First of all, I will tokenize the collection. For this I’ll consider only English alphabets (a-z) and numbers (0-9). All other characters will be discarded.
* Then, I will implement a radix tree on the generated tokens (only unique tokens will be considered). The radix tree will be implemented inside a java servlet and the servlet will be deployed on the server.
* Whenever the server will start, it will automatically build the radix tree and load it into the memory.
* When the user will start typing on the front-end of the application, the typed data will be sent to the server in JSON format. The servlet will process the request by traversing the radix tree and extracting the suggested words.
* The servlet will send the suggested words to the front-end of the application in JSON format where AJAX will be used to update the user-interface without reloading the page.

**TRIE**

***A trie is basically an ordered multi-way tree, with each node representing a character as illustrated in the figure below. Words are paths along this tree and the root node has no characters associated with it.***



***Description:***

* Root is empty.
* Every other node contains a character.
* Every leaf node contains an extra field representing a word.
* To reach a word (leaf), we have to traverse the tree from the root following the path according to the characters of that word.

***Usage for autocompletion:***

* Traverse the tree according to the initial characters entered by the user.
* Assume that we have reached a node “n” by traversing the initial characters.
* To provide the list of suggested words, retrieve all the leaves of the sub-tree rooted at “n”.

***The Trie Abstract Data Type (in Java):***

private class Trie {

private TrieNode root;

private void insert(); // called only at the time of trie-construction

private void search(String w); // main operation

}

private class TrieNode {

private char data; // the character represented by the current node

private String[] words; // for leaf nodes

private TrieNode[] children; // 0-36, considering only a-z and 0-9 characters

}

***Time Complexity Analysis:***

**insert():** Let’s always take into account the worst case timing first and later convince ourselves of the practical timings. For every Node in the Trie we had something called as children where the children is an Array. Therefore, the order of whatever operation we perform over that will be in O(1) time.

Having these in mind, for inserting a word of length 'k' we need k comparisons. By applying the Big O notation it becomes O(k) which will be again **O(1)**. Thus insert operations are performed in constant time irrespective of the length of the input string (this might look like an understatement, but if we make the length of the

input string a worst case maximum, this sentence holds true).

**search():** Same holds true for the search operation as well. The search operation exactly performs the way the insert does and its order is O(k) = **O(1)**.

***Space Complexity Analysis:***

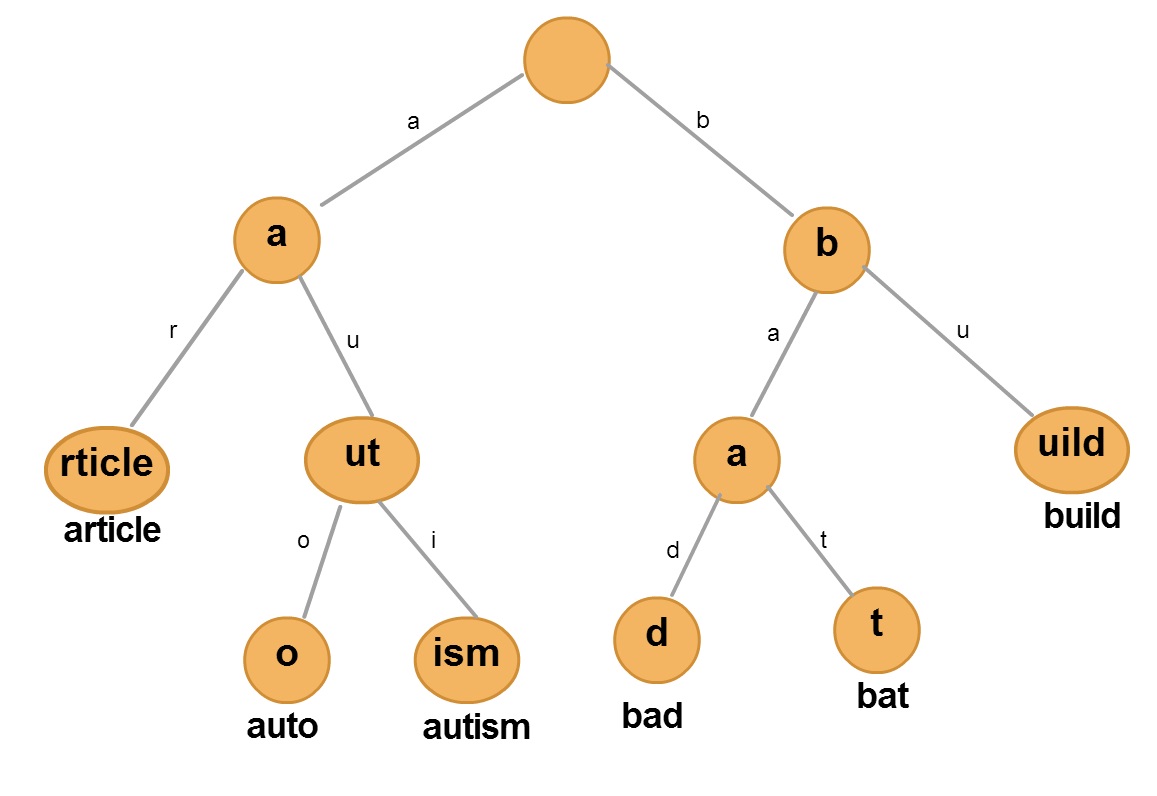
The number of nodes in our trie will be equal to the sum of the number of characters in all the unique terms of the collection. However, for words that have same prefixes, each character of the longest common prefix is represented by exactly one node.

*For large collections, such a space complexity is undesirable. Therefore, I am going to implement a compressed trie (radix tree).*

**RADIX TREE:**

***A radix tree (also called Patricia trie, Radix trie or Compact Prefix tree) is a data structure that represents a space-optimized trie in which each node with only one child is merged with its parent.***

***The previous trie example can be converted into a radix tree as follows:***

****

***Description:***

* Root is empty.
* Every other node contains a sequence of characters (string).
* Every leaf node contains an extra field representing a word.
* To reach a word (leaf), we have to traverse the tree from the root following the path according to the characters / prefixes of that word.

***Usage for autocompletion:***

* Traverse the tree according to the initial characters entered by the user.
* Assume that we have reached a node “n” by traversing the initial characters / prefix.
* To provide the list of suggested words, retrieve all the leaves of the sub-tree rooted at “n”.

***The Radix Tree Abstract Data Type (in Java):***

private class RadixTree {

private RadixTreeNode root;

private void insert(); // called only at the time of tree-construction

private void search(String w); // main operation

}

private class RadixTreeNode {

private String data; // the sub-string represented by the current node

private String[] words; // for leaf nodes

private RadixTreeNode[] children;

// 0-36, considering only a-z and 0-9 characters

}

***Time Complexity Analysis:***

**insert():** Let’s always take into account the worst case timing first and later convince ourselves of the practical timings. For every Node in the RadixTree we had something called as children where the children is an Array. Therefore, the order of whatever operation we perform over that will be in O(1) time.

Having these in mind, for inserting a word of length 'k' we need k comparisons. By applying the Big O notation it becomes O(k) which will be again **O(1)**. Thus insert operations are performed in constant time irrespective of the length of the input string (this might look like an understatement, but if we make the length of the input string a worst case maximum, this sentence holds true).

**search():** Same holds true for the search operation as well. The search operation exactly performs the way the insert does and its order is O(k) = **O(1)**.

***Space Complexity Analysis:***

The worst-case space complexity of Radix Tree is still the same as a Trie. But practically, it is quite less because a large number of English words share common prefixes.

**TOOLS, TECHNIQUES AND TECHNOLOGIES USED:**

*The following tools / techniques / technologies will be used to develop the project:*

* Java Servlets
* AJAX
* HTML / JavaScript
* Apache Tomcat Server
* Radix Tree Data Structure

**DATASET**

***I am using the Wikipedia corpus for my project.***

***About Wikipedia corpus:***

* This corpus contains the full text of the English version of Wikipedia.
* It contains 1.9 billion words in more than 4.4 million articles.

***Reasons for selecting Wikipedia corpus:***

* There are mostly “valid English words” in the collection.
* My system works best when used on valid English words.
* This is because a large number of English words share common prefixes, which gets compressed in radix trees. Moreover, average length of English words is about 6. Therefore, the average height of my radix tree will be 6. Hence, faster query processing and less space usage.
* There is a lot of variety in terms of word distributions in the Wikipedia corpus. It would facilitate exhaustive testing of the system.

**EVALUATION METRICS**

* The main evaluation metric for my system is the amount of time it takes to process the queries (i.e. provide suggestions).
* Each query will be processed in milliseconds.
* The way to evaluate query-processing time is mostly intuitive. The user will have to generate queries, and then intuitively figure out the performance of the system in terms of time taken in getting the results.
* The second metric is correctness.
* The system will have to produce the correct results. The query issued by the user (entered characters) will have to be the prefix of all the suggested words generated by the system.
* The third metric is usability. The interface will have to be simple.
* The results should be loaded on-the-fly, without refreshing the page.
* The user should not be allowed to do much work, except typing-in his queries.

***Robustness:*** Robustness is also an important metric for my project. This system will work successfully in all modern desktop-based browsers, including (but not limited to) Google Chrome, Mozilla Firefox, Internet Explorer, etc.

If time permits, I’ll try to include mobile browsers as well.

**ASSUMPTIONS / SPECIAL CASES**

* The system doesn’t discriminate the upper-case and lower-case letters. E.g. ‘a’ and ‘A’ are considered to be same. The suggestions will be in lower-case letters.
* The number of suggested words shown to the user will be limited to what could be accumulated on the computer-screen. There is no ranking used, therefore the choice of which words to display (in case all the results can’t be accumulated on the screen) will be random (but always correct).
* This is an ***exact autocomplete system***. The suggestions shown to the user will have the prefix exactly the same as the query that the user used. There is no use of Thesaurus. Therefore, approximate autocompletion is not supported.

**QUERIES TO BE USED FOR EVALUATION (TEST CASES)**

*The following test cases can be used for evaluation (it is not an exhaustive list):*

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Query** | **Valid / Invalid** | **Expected Suggestions** |
| 1 | wiki | Valid | All words having prefix “wiki”, like “wikipedia” |
| 2 | articl | Valid | All words having prefix “articl”, like “article” |
| 3 | copy | Valid | All words having prefix “copy”, like “copyright” |
| 4 | encycl | Valid | All words having prefix “encycl”, like “encyclopedia” |
| 5 | Fre | Valid | All words having prefix “fre”, like “free” |
| 6 | fre encycl | Valid | In the above case if the first suggestion was “free”. The expected suggestion in this case will be “free encycl\*” where “encycl\*” refers to all words having prefix “encycl” like “encyclopedia” |
| 7 | leag | Valid | All words having prefix “leag”, like “league” |
| 8 | 199 | Valid | All words having prefix “199”, like “1998” |
| 9 | compet | Valid | All words having prefix “compet”, like “competition” |
| 10 | A | Valid | All words that start with “a”, like “as”, “an” |
| 11 | Rive | Valid | All words having prefix “rive”, like “rivers” |
| 12 | rive of rom | Valid | In the above case if the first suggestion was “rivers”. The expected suggestion in this case will be “rivers of rom\*” where “rom\*” refers to all words having prefix “rom” |
| 13 | \_)0+ | Invalid | No suggestion. But the system stays alive (no error) |
| 14 | :P{:PI) | Invalid | No suggestion. But the system stays alive (no error) |
| 15 | @artic | Invalid | No suggestion. But the system stays alive (no error) |

**PROGRESS**

***The following has already been achieved:***

1. I have already read and understood the literature behind the proposed system.
2. Written the code for parsing the Wikipedia dataset to generate the tokens.
3. Implemented the Radix Tree data structure that takes as input the tokens of Wikipedia corpus and generates a Radix Tree.
4. The Tomcat Server is up and running.

***Remaining work:***

1. Implementing the main Java Servlet that would process the queries.
2. The GUI (including the HTML, JavaScript and AJAX).
3. Optimizing the system to make it even faster.
4. Testing.

***Difficulties faced so far:*** No major difficulty faced so far.

**REFERENCES**

* Efficient fuzzy full-text type-ahead search. Li, Ji, Li, and Feng. VLDB Journal 20(4), 2011.
* <https://creately.com/> (used for creating the tree diagrams in this document)
* <http://en.wikipedia.org/wiki/Trie/>
* <http://en.wikipedia.org/wiki/Radix_tree/>
* <https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/>
* <http://www.toptal.com/java/the-trie-a-neglected-data-structure/>