BATSMo

The Becca Alex Tierra Shruti - Model

An autocomplete analyzer mobilized for you to drive. Like the Batsmobile, only better.

BATSMo uses the complexity of a Trie Tree to predict appropriate suggestions to autocomplete user prefixes entered as input

Executive Summary

Introduction to our Project

BATSMo model: an autocomplete task

Autocomplete is a Java application that speeds up human-computer interaction by predicting and suggesting the complete word or phrase the user intends to type. Historically, a major goal of applications that display word prediction lists was to help those with physical disabilities type more quickly. Today, such applications can also be used to help people, in general, type text messages, emails and other correspondences with greater speed. This document outlines how autocomplete will be implemented using the composite design pattern and the trie and HashMap data structures. Finally, this document includes a UML diagram containing the six classes used in the software's implementation. It also includes three interfaces: IAutocomplete, which specifies the program's operations; ITrie, which lists the operations of the trie data structure, and IFileReader, which outlines the functionality of the FileReader class.

Report Features

- UML diagram containing the six classes used in the software's implementation.

Three interfaces:

- IAutocomplete, which specifies the program's operations;
- ITrie, which lists the operations of the trie data structure.
- IFileReader, which outlines the functionality of the FileReader class.

Project Description

Overall Process Flow

To start the application, the user must provide a name of the text file from which our auto-compiler will learn. Running the program will initiate a myriad of steps that will ultimately return a list of suggested recommendations by our Model. On the user side, the user will have the option of interacting with a Graphical User Interface (GUI) taking input from command line or via the command line only..

Inside the Main Program

Int nSuggest = 5 // how many suggestions do we want to offer?
fReader = new FileReader ("filename.txt") ;
HashMap<String, Integer> freqTable = fReader.getWords() ;
HashMap<String, HashMap<String, Integer>> successors = fReader.getSuccessors() ;
Autocomplete a = new Autocomplete (freqTable, successors) ;
Within the autocomplete constructor the following tasks are carried out:
build tries ;
save freqTable to wordTable ;
initialize successorMap: call buildSuccessorMap(successors)

Note:
buildSuccessorMap() will do something like
for each key k in successors:

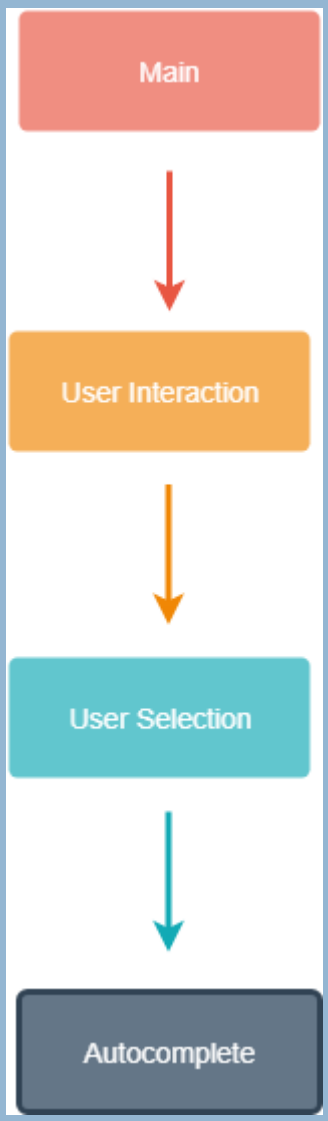
HashMap<String, Int> h = k.value ;
PrefixPair[] aSucc = convertToPrefixPair(h) ;
PrefixPair[] sortedSucsrs = sortLixicographically(aSucc) ;
successorMap.add(k, sortedSucsrs) ;

User Interaction

Input = everything that is typed to the input line of UI
first user input: user types first letter (or a few with no space) for the first time
String input = <what was typed>
Array<String> candidates = a.allPredictions(input);

inside Autocomplete allPredictions(input) function:
tokenize input (by space/./,/ etc)
at this time input only has one word, call it "word", so previous is null
isWord returns false, so call getMatchSuggestions(word, null, nSuggest)
it will go to tries array to find the root corresponding to first char of input
let's call it tries[i]

Process Flow



Project Description Cont'd

call tries[i].search(input) to find the node (name it "n") containing the whole input
get all words that starts from this input by calling tries[i].listOfPredictions(n)
sort the returned list<strings> "L" using data from wordTable calling sortMatchesByFreq(L)
return the sorted collection of candidates

// *user chooses one of our suggested words*
We add a space character to it, so input = "choosenWord" + ' ' ;
Do the usual routine to find the successor of the last input word (which is in this case "choosenWord")
candidates = a.allPredictions(input) ;
inside Autocomplete allPredictions(input) function:
isWord returns true, so call getSuccessorSuggestions (input - ' ')
that goes into successorMap, gets Array of PrefixPairs, calls
sortSuccessorsByFreq() on this array, and returns sorted collection of strings

// *user types the word and enters space* - Behavior is almost same as in previous case.
// *user starts typing 3rd word*, typed a letter a few (no space)
Input string now has "word1 word2 prefix"
We need to look for this prefix in list of suggested followers of the word2 first; if nothing is found (or less than we want to output), look inside the tries[] as usual.
We don't need word1, only the prefix and the word before it, so we can discard everything before.
Input = "prefix" ;
PreviousWord = "word2"
candidates = a.allPredictions(input) ;
inside Autocomplete allPredictions(input) function:
looking at the last character, isWord() returns false
tokenize the input and get last two words: "prev" and "prefix"
call getSuccessorSuggestions (prev) that will return PrefixPairs[] (call it prevMatches)
call binarySearch(prefix, PrefixPair[] prevMatches) to get only those matches that start from prefix, that will return PrefixPairs[] (call it prevPrxMatches)
call sortSuccessorsByFreq(prevPrxMatches): to sort them by frequency that will return array of strings (call it array1)
if length of this array is less than what we want to return (l < nSuggest)
call getMatchSuggestions(prefix, array1, nSuggest-array1.length)
that will return String[] (call it array2) of size nSuggest-array1.length
Now we need to make a new array of strings of size nSuggest, copy array1, then copy array2, and return it.

The Composite Design Pattern

BATSMo Autocomplete Trie

Composite design pattern is the most suitable for our implementation. By definition the composite design pattern is meant to "compose" objects into tree structures to represent part-whole hierarchies. It allows to have a tree structure and ask each node in the tree structure to perform a task. This is perfect for our autocomplete task that is using a Trie Tree structure to predict prefix pairs for any input (adopted from the word retrieval, because we are implementing and information retrieval task) .

The Composite Pattern has four participants:

Component (BATSMo Interface)

Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.

Leaf(Trie Class)

Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition. The Trie Class leaf will be a pointer to a hashmap of frequencies that will predict suggestions for our autocomplete.

Composite(Trie Class)

Composite stores child components and implements child related operations in the component interface. We will have a Trie Class that stores individual data information and next pointer to be utilized in a tree. This is the quintessential design pattern for building a Trie Tree and that it is a composite of Trie Nodes.

Client(GUI)

Client manipulates the objects in the composition through the component interface.

Composite Design

What makes a program a good candidate for a composite design?

- Component – central interface
- Leaf – primitive parts to a whole
- Composite – Component interfaces representing typically a tree structure
- Client – an interface typically representing by a GUI

Moreover, this design pattern was chosen because it's most suitable in a task where clients can ignore the difference between composition of objects and the individual objects themselves (aggregations of objects). We will be using code stored in TrieNodes nearly identically, using multiples of the nodes in the same way, and using those individual objects to build a more complex structure.

Data Structures

An introduction to our interfaces

Trie Trees

We create a trie to store strings seen in the dataset and past searches by the user. The user is provided with a list of recommendations to select from and autcomplete the prefix entered based on strings stored in the Trie. Every TrieNode has a root associated with it.

Tries are different from binary trees in that each node can have an arbitrary number of children, and rather than having named pointers to children, the pointers are the values in a key-value map. All the descendants of a node have a common prefix of the string associated with that node.

We've designed our trie so that each trie node consists of a value which represents a letter of the english alphabet and the letter of the current node (children having 'a'-'z' characters). So we have an array of 26 TrieNodes as its children. It also has a Boolean variable associated with it to detect that the node represents the end of a word.

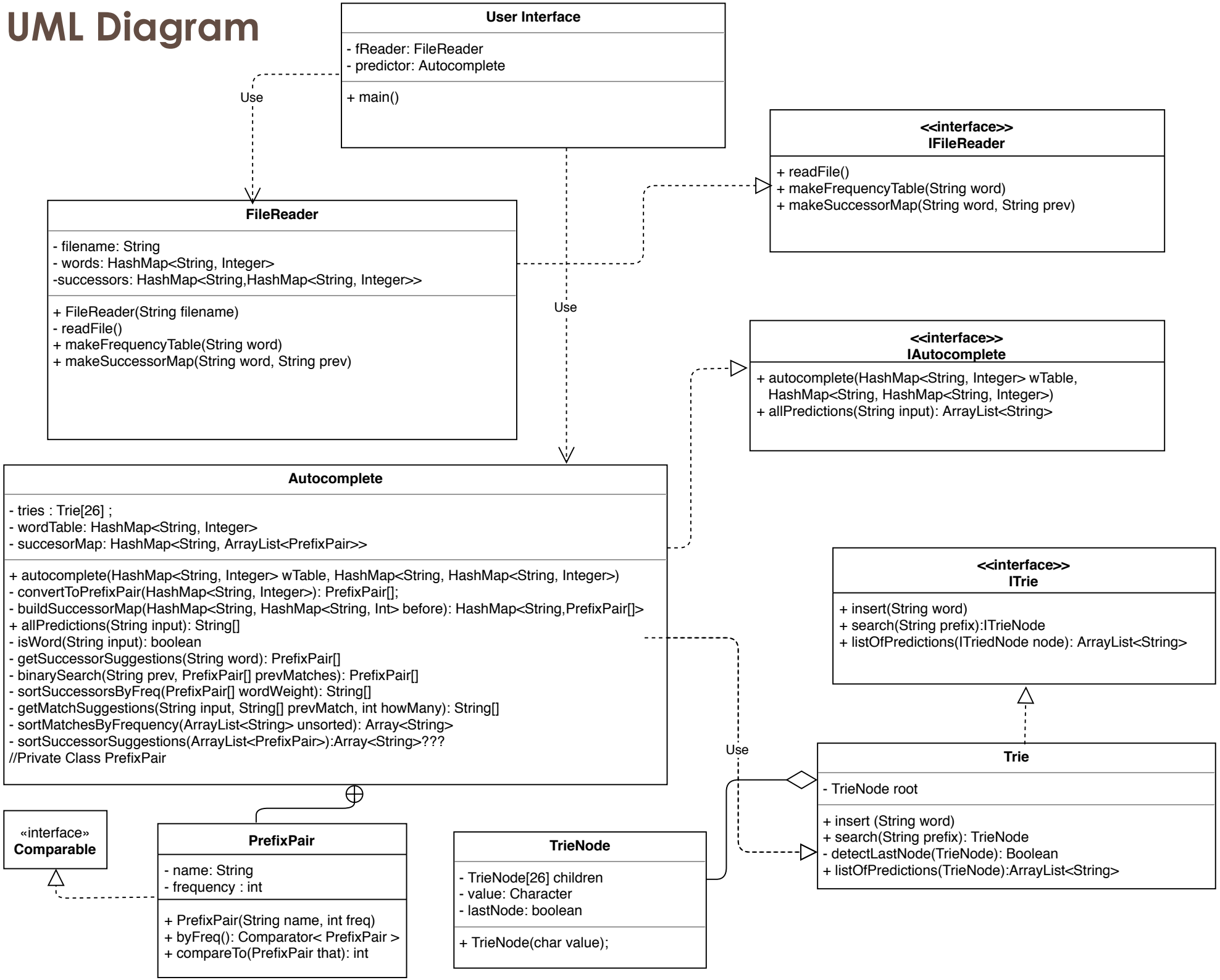
HashMap

We use the number of times a word appears in a given dataset as our main metric in ranking suggestions given to the user.
Another HashMap, mapping a word to its possible successors is created using a HashMap of HashMaps. The top 5 commonly occurring successors are stored in an arraylist, sorted based on their frequency from the HashMap.
Additional: Array: An array of size 26 is created to store Tries representing each letter in the alphabet.

Data Structures Used

- Trie Trees
- HashMap.

UML Diagram


User Interface
- fReader: FileReader
- predictor: Autocomplete
+ main()
Use
<<interface>>
IFileReader
+ readFile()
+ makeFrequencyTable(String word)
+ makeSuccessorMap(String word, String prev)
FileReader
- filename: String
- words: HashMap<String, Integer>
-successors: HashMap<String,HashMap<String, Integer>>
+ FileReader(String filename)
- readFile()
+ makeFrequencyTable(String word)
+ makeSuccessorMap(String word, String prev)
Use
<<interface>>
IAutocomplete
+ autocomplete(HashMap<String, Integer> wTable,
HashMap<String, HashMap<String, Integer>)
+ allPredictions(String input): ArrayList<String>
Autocomplete
- tries : Trie[26] ;
- wordTable: HashMap<String, Integer>
- succesorMap: HashMap<String, ArrayList<PrefixPair>>
+ autocomplete(HashMap<String, Integer> wTable, HashMap<String, HashMap<String, Integer>)
- convertToPrefixPair(HashMap<String, Integer>): PrefixPair[];
- buildSuccessorMap(HashMap<String, HashMap<String, Int> before): HashMap<String,PrefixPair[]>
+ allPredictions(String input): String[]
- isWord(String input): boolean
- getSuccessorSuggestions(String word): PrefixPair[]
- binarySearch(String prev, PrefixPair[] prevMatches): PrefixPair[]
- sortSuccessorsByFreq(PrefixPair[] wordWeight): String[]
- getMatchSuggestions(String input, String[] prevMatch, int howMany): String[]
- sortMatchesByFrequency(ArrayList<String> unsorted): Array<String>
- sortSuccessorSuggestions(ArrayList<PrefixPair>):Array<String>???
//Private Class PrefixPair
<<interface>>
ITrie
+ insert(String word)
+ search(String prefix):ITrieNode
+ listOfPredictions(ITriedNode node): ArrayList<String>
Use
Trie
- TrieNode root
+ insert (String word)
+ search(String prefix): TrieNode
- detectLastNode(TrieNode): Boolean
+ listOfPredictions(TrieNode):ArrayList<String>
«interface»
Comparable
PrefixPair
- name: String
- frequency : int
+ PrefixPair(String name, int freq)
+ byFreq(): Comparator< PrefixPair >
+ compareTo(PrefixPair that): int
TrieNode
- TrieNode[26] children
- value: Character
- lastNode: boolean
+ TrieNode(char value);