Prefix Trees (Tries) for Tamil Language Processing

Elango Cheran
July 2017

Enabling Tamil language computing on each new technology platform requires ensuring that each layer of the technology stack supports the language. While it is useful to assess what those layers are, and the progress that has been made for Tamil over the years, it is also important to look forward to solving future problems that need work at higher-level layers. Towards that goal, the prefix tree (trie) is an important data structure that can be used to enable basic Tamil language operations that enable more advanced work for Tamil to be done. The ways in which we can apply prefix trees for Tamil are general enough that they would very likely apply to other Indic languages, too.

A prefix tree is a special type of tree data structure that is used to efficiently store several strings that may share various prefix substrings in common with each other. Each letter of a string is stored as a node, with each subsequent letter stored in a child node of the previous letter's node. For example, if we constructed a prefix tree to hold the strings [bot, bow, be, bed, go, got], it would look like:

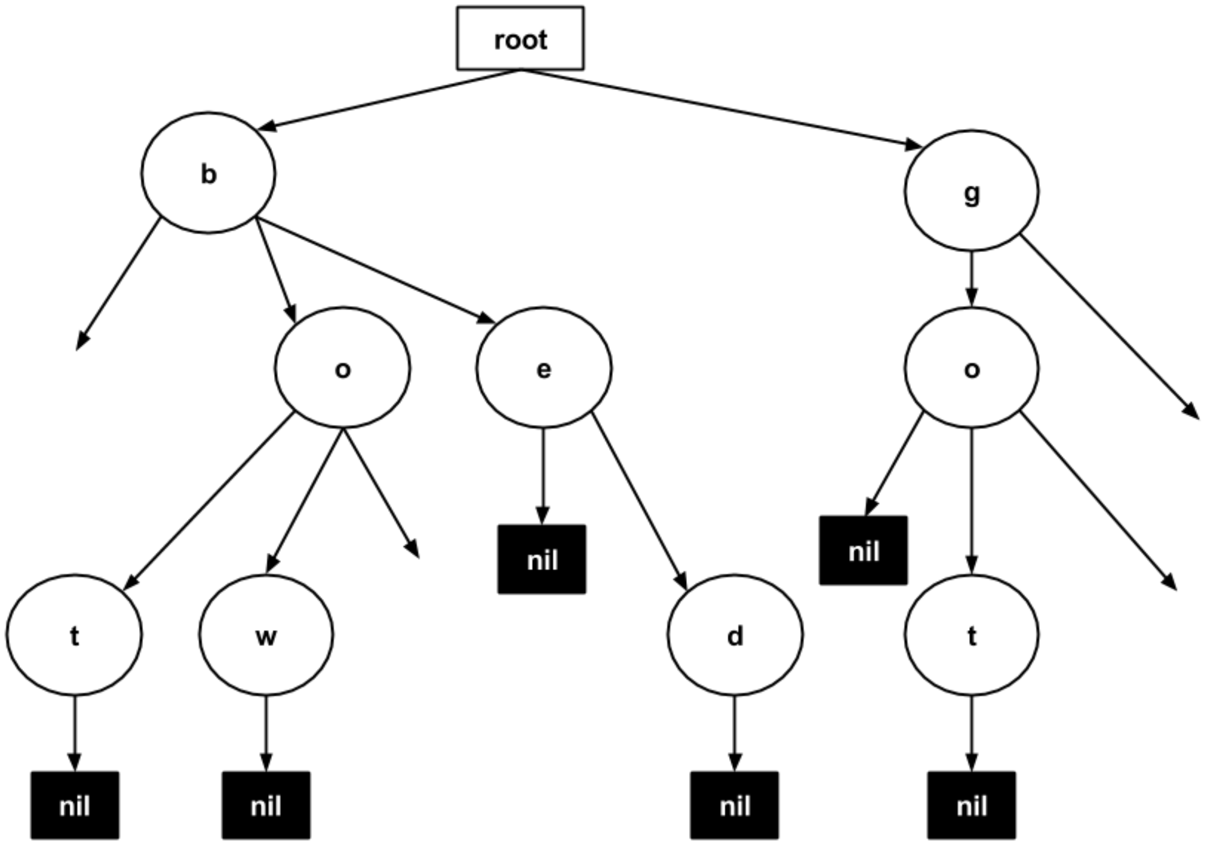

Figure 1: a prefix tree constructed from a list of strings

More generically, prefix trees can contain sequences made up of (a set of) elements. Most commonly, prefix trees are used to represent strings, and strings are merely sequences of characters. Following each path from the root of a prefix tree to a leaf node will contain the ordered elements of an input sequence. In Figure 1, a leaf node is represented as "nil", which can be considered like an end-of-word marker that is not a part of the tree's input sequences.

There are some easily solvable challenges in the basic processing operations of Tamil text for which prefix trees offer a natural solution. To explain those challenges, let's first examine the English example in Figure 1. In the Unicode character set, every English letter is represented by a single Unicode codepoint in the specification. In a programming language like Java which supports Unicode, these codepoints each map to a single Character value, where the Character refers to the data type as provided by the programming language. So an implementation of a prefix tree that only supports English could have each internal tree node hold a single Character value. The Unicode specification for Tamil (and other Indic languages) represents the logical letters of the alphabet sometimes with more than codepoint. Letters like அ..ஔ (vowels, or உயிரெழுத்து) and க..ன (consonant+"a", or அகரமெய்யெழ்த்து) are all represented by one codepoint. Letters like க் (consonant, மெய்யெழுத்து) and கா..கௌ (C+V except C+அ, அகரமெய்யெழுத்து தவிர மெய்யெழுத்து) are represented by two successive codepoints in Unicode. See Figure 2. In the end, there is a one-to-one correspondence between a Tamil Unicode character sequence and the logical Tamil text that it creates, and that is the ultimate goal of any universal language encoding specification. Any challenges to deal with the text thereafter at a higher level are technical ones for software developers.

Text	**Logical letters**	**Number of logical letters**	**Text Unicode codepoints**	**Number of Text Unicode codepoints**
go	g, o	2	g, o	2
got	g, o, t	3	g, o, t	3
bot	b, o, t	3	b, o, t	3
bow	b, o, w	3	b, o, w	3
தணி	த, ணி	2	த, ண, ி (0BBF)	3
தணிகை	த, ணி, கை	3	த, ண, ி (0BBF), க, ை (0BC8)	5
வருகை	வ, ரு, கை	3	வ, ர, ு (0BC1), க, ை (0BC8)	5
வருக	வ, ரு, க	3	வ, ர, ு (0BC1), க	4

Figure 2: a list of words, letters, and Unicode points for English and Tamil

The above description of how to map Tamil language letters into the corresponding Unicode codepoint(s) already reflects the challenge somewhat. Figure 2 shows not only the difference between number of logical letters and codepoints, but it also shows the counter-intuitive property that the character sequence of வருக is a subset of வருகை. For these reasons, one of the first tasks that naturally arises when dealing with Tamil text in Unicode is to convert the character sequence into a sequence of strings representing the logical letters. This parsing task, like any other stateful task involving transitions between states, can be represented by a finite state machine (FSM). The FSM represents all of the logic used to enumerate the states and determine the conditions required in order to transition from one particular state to another. Parsers naturally can be described using FSMs, and in the case of parsing Tamil text, the FSM containing all valid transition paths is the prefix tree itself. Thus, the prefix tree containing the string representations of all logical Tamil letters is the simplest mechanism for writing code to parse Tamil text (Figure 3).

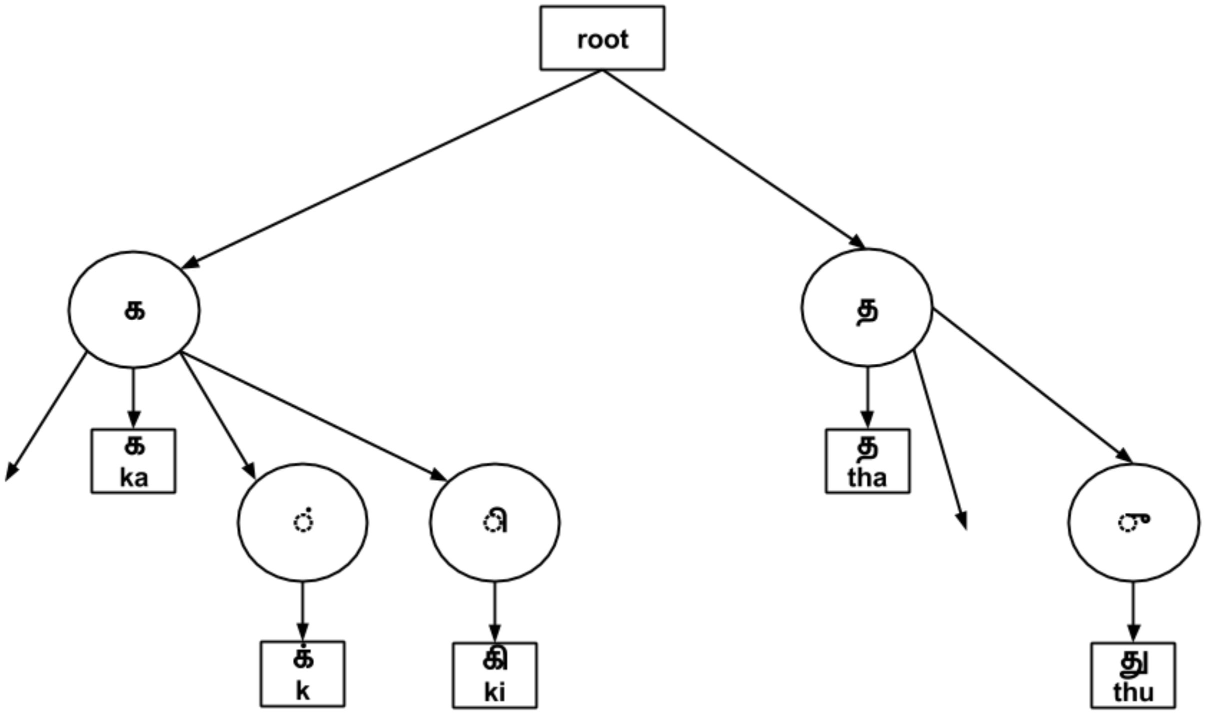

Figure 3: a prefix tree containing each Tamil letter's character sequence representation

For example, a Tamil text-to-letter parsers written in more imperative code will require complex logic for consonants since there must be a peek ahead at the next character (if there is one) to distinguish அகரமெய்யெழுத்து (C+அ) from மெய்யெழுத்து (C) or any other உயிர்மெய்யெழுத்து (C+V). In the case of வருகை, after parsing the வ, the next character in the stream is ர. Until we look ahead at the next character, we won't know if ர is the full letter (in the case of the word வரவு) or if the next character should be combined for the full letter (in the case of வருகை, where the next character after ர is ு (0BC1)). The imperative style implementation also requires verbose nested if-else statement logic. However, a prefix tree offers the operation of longest shared prefix, meaning that given an input string, it will return the longest string in the prefix tree shared with the input. In the English example, with an input of “gothic”, the prefix tree will return “got”, even though “go” exists in the tree and is also a prefix. This operation is exactly all that is needed for Tamil text-to-letter parsing. See Figure 4:

Input string	**Output from Tamil letter prefix tree’s longest prefix function**
வருகை	வ
ருகை	ரு
கை	கை

Figure 4: the input and output of a prefix tree’s longest shared prefix function

The discrepancy in approaches is more apparent for letters with longer Unicode codepoint sequences. Some Grantha letters in Tamil text require up to 4 codepoints. An imperative style parser code may need a 4-level nested if-else block, but the prefix tree-based parser code remains unchanged. Dealing with letters requiring more than 2 characters is an infrequent case for Tamil text, but it is perhaps relevant and significant for other Indic languages.

It is important to take note that many useful Tamil language operations do not happen at the unit of a letter (எழுத்து) but instead at unit of a phoneme (ஒலியன்). Tamil, as an agglutinative language, encodes much grammatical information through word suffixes. Suffixes (விகுதி) are so common that the basic rules governing word changes when adding suffixes are given a name (சந்தி, “sandhi” - to meet). A simple case to show the importance of phoneme units over letter units for grammar would be to indicate “and” for a compound subject, indicated by adding “-உம்" to each subject. For the words மாமா and மாமி, the sandhi rules imply the changes மாமா + (வ்) + உம் = மாமாவும் and மாமி + (ய்) + உம் = மாமியும், resulting in “மாமாவும் மாமியும்". The inserted வ் and ய் is based on the vowel sound of the final letter in both words, not the final letter’s consonant sound. The words அக்கா and அண்ணா also add வ், whereas தங்கை and தம்பி add (ய்).

Word	**Letters**	**Final letter**	**Phonemes**	**Final phoneme**	**Sandhi change (-உம்)**
மாமா	மா, மா	மா	ம், ஆ, ம், ஆ	ஆ	வ்
மாமி	மா, மி	மி	ம், ஆ, ம், இ	இ	ய்
அக்கா	அ, க், கா	கா	அ, க், க், ஆ	ஆ	வ்
தங்கை	த, ங், கை	கை	த், அ, ங், க், ஐ	ஐ	ய்
அண்ணா	அ, ண், ணா	ணா	அ, ண், ண், ஆ	ஆ	வ்
தம்பி	த, ம், பி	பி	த், அ, ம், ப், இ	இ	ய்

Figure 5: letters vs. phonemes for Tamil words

There are extra sandhi rules applied in the case of noun case suffixes (வேற்றுமை). For two words with the same last letter, மடு and காடு, adding the same case suffix “-இல்" operates differently. For மடு, the change is simpler: மடு + (வ்) + இல் = மடுவில். For காடு, an arithmetic on the word happens first: காடு + -இல் = (க், ஆ, ட், உ) - உ + ட் + (இ, ல்) = காட்டில். Because of the final -ட், the final -உ is dropped, the -ட் is doubled, and then the இல் is added.

We can use a prefix tree to split a Tamil word into phonemes by modifying the tree to allow a value to be associated with each leaf node (input string), much like a map/dictionary. Each Tamil letter string in the tree is associated with its phoneme sequence: கி -> [க், இ]; கூ -> [க், ஊ]; க் -> [க்] (Figure 6). Linguistic operations in Tamil often operate on a sequence of phonemes and return a sequence of phonemes (Figure 7). Given phonemes, a prefix tree can be created that converts the phoneme sequence back into regular Tamil text (Figure 8).

Input string	**Output from Tamil letter prefix tree's associated phoneme sequence**
காடு	[க், ஆ]
டு	[ட், உ]

Figure 6: the input and output of a Tamil letter prefix tree modified to be associative

```
(வரையறு-செயல்கூறு வேற்றுமை-முன்-மாற்றம்
 [சொல்]

 (வைத்துக்கொள் [
         எழுத்துகள் (தொடை->எழுத்துகள் சொல்)
          ஒலியன்கள் (தொடை->ஒலியன்கள் சொல்)
          கஎ (கடைசி எழுத்துகள்)
          கஒ (கடைசி ஒலியன்கள்)]

  (பொறுத்து
   ...
   (= "டு" கஎ)
   (செயல்படுத்து தொடை (தொடு (கடைசியின்றி எழுத்துகள்) ["ட்ட்"]))

   (= "று" கஎ)
   (செயல்படுத்து தொடை (தொடு (கடைசியின்றி எழுத்துகள்) ["ற்ற்"]))

   :அன்றி
   சொல்)))
```

Figure 7: a function to prepare a noun for adding a case suffix (வேற்றுமை)

Input string	**Output from inverse Tamil phoneme prefix tree’s associated letter string**
க்ஆட்ட்இல்	கா
ட்ட்இல்	ட்
ட்இல்	டி
ல்	ல்

Figure 8: the input and output of a prefix tree constructed as the inverse of Figure 6’s tree

Using prefix trees that have been modified to be associative, we can easily define conversions from Tamil text to the old pre-Unicode encodings (and vice versa). More importantly, once we start to think of Tamil text less in terms of the underlying character sequences and more as sequences of logical letters or logical phonemes, implementing other operations becomes clearer. For example, true lexicographical sorting can be achieved for Tamil by splitting a word into letters and combining with a lookup map that indicates the relative ordering of each letter. If we define the lexicographical ordering of Tamil letters as [அ, ஆ, …, ஃ, க், க, கா, …, கௌ, ங், ங, …, ன், ன, …, னௌ], our lookup map would be {அ 0, ஆ 1, …, ஃ 12, க் 13, க 14, கா 15, …, கௌ 25, ங் 26, ங 27, …, ன் 235, ன 236, …, னௌ 247}. Then sorting Tamil words becomes equivalent to sorting sequences of numbers, which is straightforward. In a sense, sorting sequences of numbers is equivalent to sorting English text because of the one-to-one mapping of English letters and Unicode codepoints.

Once we use the appropriate data structures to model our domain more accurately, the functions we need to solve our basic problems become clear, and we can begin to solve more advanced problems. For example, an intelligent spell checker might use sequence alignment to measure closeness, for which the 2 most basic methods are global (Needleman-Wunsch) and local (Smith-Waterman). Using the phoneme representation of strings would not only fit the algorithms’ designs, but it would also provide better results. There is much room to explore the implications of a phoneme-based modeling of Tamil text (as is done in Korean and other languages), but prefix trees offer a necessary first step in that direction, and they are general enough to be applicable to other Indic languages, if not more.

An open-source library that implements these ideas, along with example projects in Java and JavaScript using the library, including the above code, at: https://github.com/echeran/clj-thamil