Discovering Affixes Automatically

Claudia Borg

November 16, 2016

1 Introduction

One of the components in computational morphology that we will delve into during this course is that of automatically acquiring 'morphological' information. In this assignment, we will be looking specifically at establishing whether some string sequence can be considered an affix or not by using transitional probability. In order to facilitate this task, we will be using a data structure called a Trie[1] (pronounced "tree", from the word re-trie-val, or "try", for differentiation), which provides us with an efficient implementation both in terms of storage of data and as well quick access of data for our queries.

The Trie is a type of tree structure whereby each node represents a single character. A sample is shown in figure 1. The root node is a symbolic node that is used as a point of entry to the whole structure. From the root, we can then access all the different children. Each node can have several children, but each must be unique — for instance, the root node can only have one child representing the character 'a'. All words starting with 'a' will share this node. In our example below, the whole string sequence 'report' is shared by several words, and then it branches off to continue representing the remaining words, 'reporter', 'reportable', etc.

We will be using a word list, together with frequency counts, complied from a corpus[2]. This list will be in the form of a text file, with one word per line, together with its token count[3]. The first part of the assignment will focus on the basic building blocks to get the Trie data structure up and running in a rudimentary shape. The second part of the assignment will

[1]For a more comprehensive explanation of a Trie Dictionary refer to `http://en.wikipedia.org/wiki/Trie` as an initial description. Last accessed 5-July-2012

[2]Frequency lists will be given to you for both English and Maltese. You can choose to work with either one, or to try and use both. The Maltese list is taken from the Maltese Language Resource Server (MLRS) Corpus which is also available through an online interface `http://mlrs.research.um.edu.mt/index.php?page=31`

[3]Token or Frequency count refers to how many times a particular word appears in the whole corpus.

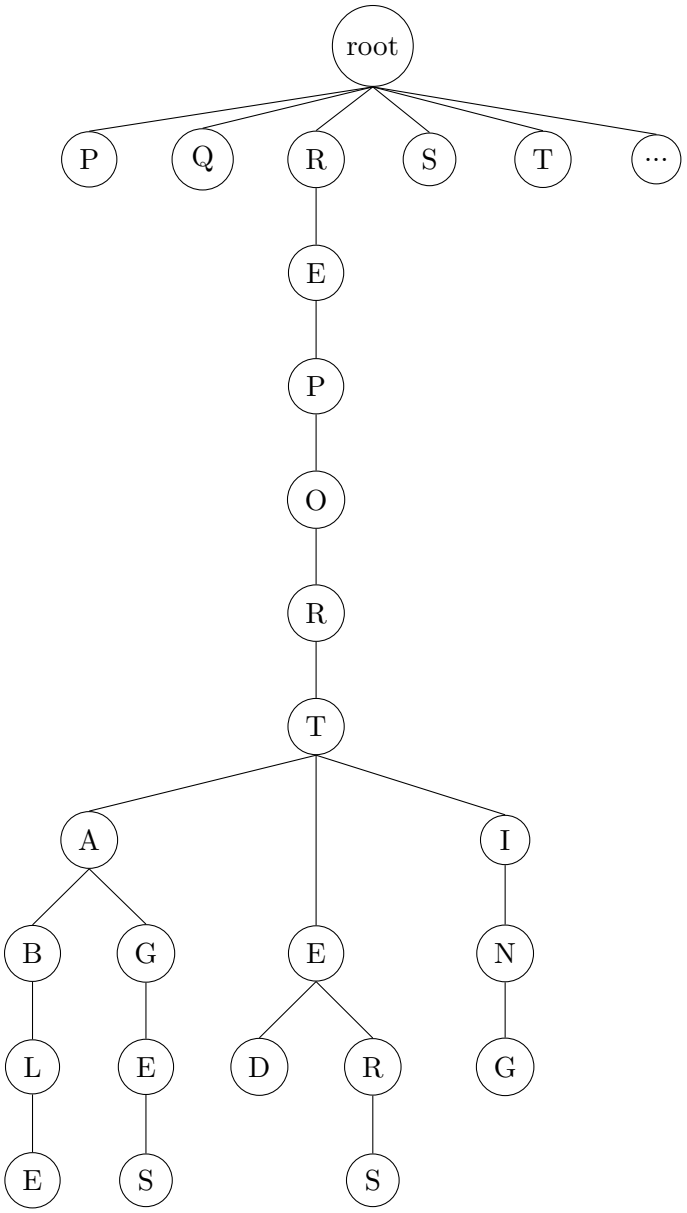

Figure 1: Trie Example — The vanilla version for the dictionary entries: report, reportable, reportage, reportages, reported, reporter, reporters, reporting

then look at the necessary extensions required in order to use transitional probability to be able to calculate potential prefixes and suffixes.

2 Trie Implementation

The first basic task will consist of implementing a 'rudimentary' Trie. There are three primary objects that are required:

1. Node — The node data structure will provide us with the 'blueprint' to create all the required nodes.
2. TrieDictionary — The Trie data structure provides us with an 'interface' to access our Trie. All requests will be done to this structure, which will have access to the nodes.
3. Main Class — From where we interact with the Trie.

General Hint: Read through Tasks 1–3 several times before starting to implement them. First build a mental model and make sure that you understand the general principles of Nodes and Tries, then scribble it on paper/s, and only then start implementing.

2.1 Task 1: Implement a simple Node structure

The first task consists of creating the appropriate Node skeleton structure. Remember that this class will act as a blueprint for the many nodes that we will need to create. Start by spending some time and analysing what different information needs to be stored in order to handle the different type of nodes:

1. Root Node — this one is special. It does not represent anything really, its like a dummy node, but we need it so that we always have one single node through which we access everything else. This is marked as the red node in figure 2.
2. Regular Node — This is our normal node. It represents one single character, and it might or might not have children. These are marked as black nodes in figure 2. There are several of them, and this is our most basic building block.
3. End Token Node — This is similar to our regular node, with the only difference that we set a marker in this node to note that the path from the root to this node represents a valid word. This is marked in a double-lined blue circle in figure 2 and you can easily make out the different words from the root to each of these blue nodes.

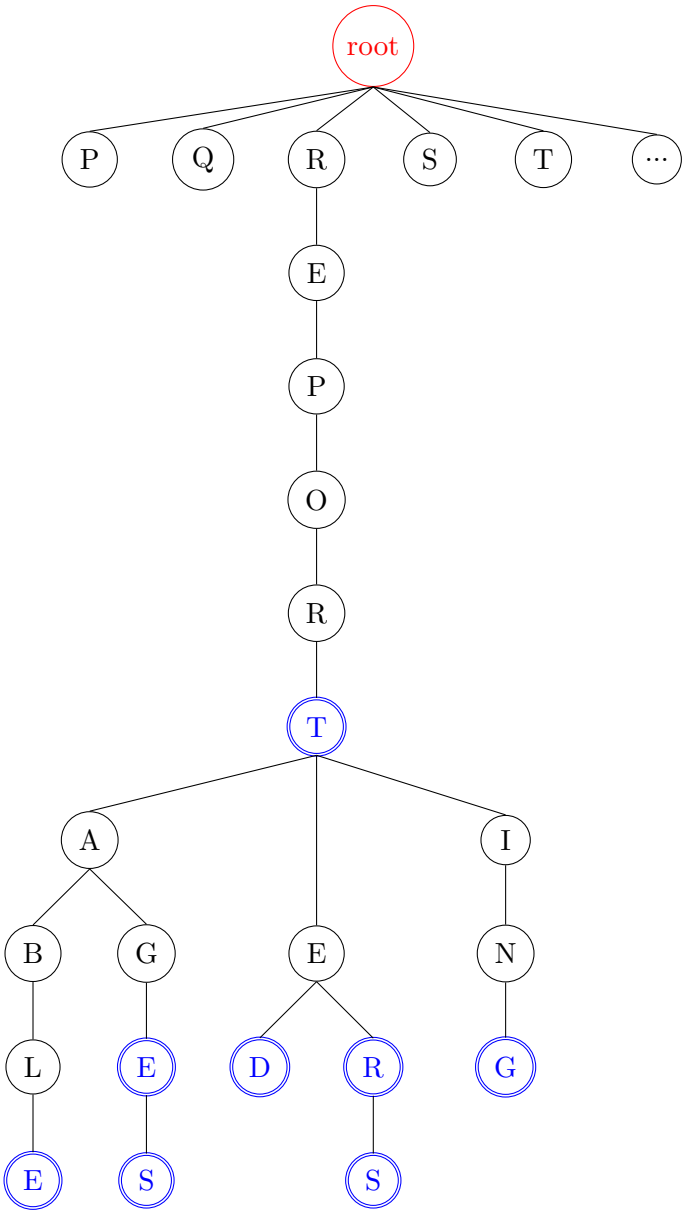

Figure 2: Trie Example — The coloured version for the dictionary entries: report, reportable, reportage, reportages, reported, reporter, reporters, reporting. The red node is the root node, the black nodes are normal nodes and the blue double-lined nodes are the end token nodes.

If we look closely to the diagram, we can reason out a few things that could help us during the implementation:

1. Only the root (i.e. the 1^{st}) node is red.
2. Black nodes always occur internally and are never found as leafs (i.e. nodes on the very edge, without any children) — why is this?
3. Blue node can occur both at internal points but also occur as end points (leafs) of the Trie — why?
4. What is the primary difference between the blue nodes and the black nodes?
5. Do we need different classes for these two nodes or can we implement one class and cover both types of nodes?
6. How about the root node — does it require special implementation, or can we get away with implementing all these three types of nodes in one?

Short answer: SINGLE node implementation

Requirement: In your documentation take care to explain clearly why only one node class is required, and demonstrate how you cater for the different node 'types' with one class.

Next item to consider: each node can have several children. How would you store these children? Remember, these are nodes as well. So in your node class, you need storage for a list of children of type `Node`.

Once you have thought of the data ('variables') that the node will hold, next you will need to think about the methods that will be required.

1. Constructor — we need to be able to create a node. We know that each node represents a character, so we can pass on this character to our constructor. However, does every single node really represent a character? What about the root node?
2. Get Node Character — this might be useful when we want to probe which character a node represents.
3. Search For Child — we will need a searching facility to either access a particular child, or to check if it exists.
4. Add an Input String — one of our main functions that will be used when a string is entered into the dictionary.

Listing 1 provides you with the basic 'code' if you had to implement this in Java. There are *holes* in the code which you are meant to come up with your own solution and technique. Make sure that you document your choices and reasoning. Feel free to change the code/variables/naming that is used. Make this your code that you are able to understand and reason about.

NOTE: This will be updated to show Python code

Listing 1: `TrieNode` Class

```
1  public class TrieNode {
2
3    char nodeChar;
4    boolean endToken = false;
5    TrieNode[] ChildNodes = new TrieNode[0];
6
7    public TrieNode() { } //empty constructor
8    public TrieNode(char c) {
9      this.nodeChar = c;
10   }
11
12   public char getNodeChar() {
13     return this.nodeChar;
14   }
15
16   public TrieNode getChildNode(char c) {
17     //search through ChildNodes[]
18     //if you find a node represented by c
19     //then return that node
20     //else return null
21     return null;
22   }
23
24   public boolean addEntry(String input) {
25     //take the first char of the input string
26     //and check whether it exists, and if not
27     //create a childNode with that char
28     //use the childNode to addEntry with
29     //the remaining of the String
30     //When you reach the last letter of the input string
31     //set endToken = true
32     //return true upon success
33     return false;
34   }
35 }
```

2.2 Task 2: Implement a Trie Dictionary Class

The second task is the basic implementation of the Trie Dictionary class. This is necessary because as programmers, we do not access the nodes directly, not even the root node — we always access the dictionary as a whole. Reflect on what type of queries we will be doing with out Trie Dictionary:

1. Constructor — create a new dictionary. This would be empty at this stage, but surely the one thing that we know is that a dictionary must have a root node in order to access the remaining of the trie. At `TrieDictionary` level, we only care about the root node. The dictionary has no other knowledge about the rest of the nodes.

2. Add a word — we want the facility to add a word to the dictionary. How? We simply tell the root node to add the word we would like it to add. How it does it? From a `TrieDictionary` level we don't care — we let the root node handle this. Remember that the root node is an instance of a `Node`, and has the facility to add an input string.

3. Query Word — we also would like the facility to check whether a word exists in our dictionary or not. This is something that we have not considered during the `Node` class implementation, however it is a very handy feature to have. The technique is simple — we ask the root node if a word exists or not. The tricky part is that we must go back to the `Node` class and include a method to search for the path that represents the query word. Remember, we know that a word exists in the dictionary when we have a path from the root node to a *blue* node — this part is implemented in the `Node` class. From a `TrieDictionary` level we only care about the final answer from the node. As a hint, implement this only when you fully understand how a word is added to the dictionary, because searching for a word is quite a similar process.

NOTE: This will be updated to show Python code

Listing 2: `TrieDictionary` Class

```
1  public class TrieDictionary {
2
3    TrieNode rootNode;
4
5    public TrieDictionary() {
6      rootNode = new TrieNode();
7    }
8
9    public boolean AddWord(String word) {
10     return rootNode.addEntry(word);
11   }
12
13   public boolean hasEntry(String word) {
14     return rootNode.hasEntry(word);
15   }
16 }
```

2.3 Task 3: Implement a Main Class

This task will simply be a way of interacting with the Trie Dictionary. We will add entries, check if entries exist, and so on. We will do this gradually in simple subtasks.

1. Create a Trie Dictionary
2. Create two lists of words — an array of 10 words in each one should be fine as an initial test. One list will be an input list to the Trie, the other list will contain different words that should not exist in the Trie.
3. Iterate through word list 1 and add each word to the Trie
4. Iterate through word list 1 and check if each word is in the Trie Dictionary
5. Iterate through word list 2 and check if each word is in the Trie Dictionary (this one shouldn't be).

The above is quite easy, and can be similar to listing 3, showing an output described in listing 4.

Requirement: What worked, what didn't? Take note of your errors and include a short description in your documentation. Did you get the expected output immediately right? What did you do to fix your problem?

The next required extension is that you are able to take the word list directly from a file. The file will contain one word per line and will have the associated token count for each word, tab delimited.

Create a function in your `Main` class that will read the word list and store it in an appropriate data structure of your preference.

1. What storage did you choose?
2. Give a short description of how this structure meets your requirements.

This is a preparation for the next section of the assignment, where apart from adding words in the dictionary, we will also need to take note of the token count amongst other things. For now, you can test your `TrieDictionary` with the full word list (ignoring token counts) — how fast is the input? Check a few entries and ensure that the behaviour is as expected.

NOTE: This will be updated to show Python code

Listing 3: `Main` Class

```
1  import java.util.ArrayList;
2
3  public class MyMain {
4
5    public static void main(String[] args) {
6      TrieDictionaryT1A myTrie = new TrieDictionaryT1A();
7      ArrayList<String> goodWords = new ArrayList<String>();
8      ArrayList<String> badWords = new ArrayList<String>();
9
10     goodWords.add("report");
11     goodWords.add("reporter");
12     goodWords.add("reporters");
13     goodWords.add("reported");
14     goodWords.add("reportable");
15     goodWords.add("reportage");
16     goodWords.add("reportages");
17     goodWords.add("reporting");
18
19     badWords.add("repo");
20     badWords.add("save");//not a bad word, but for now not part of dictionary
21     badWords.add("t");
22     badWords.add("ejis");
23     badWords.add("repe");
24
25     System.out.println("-------- Adding Words to Trie --------");
26     for (String s1 : goodWords) {
27       System.out.println("Adding " + s1 + " " + myTrie.AddWord(s1));
28     }
29     System.out.println("-------- Checking if all good words return true
           --------");
30     for (String s2 : goodWords) {
31       System.out.println("Checking for " + s2 + " " + myTrie.hasEntry(s2));
32     }
33     System.out.println("-------- Checking that all list of bad words return false
           --------");
34     for (String s3 : badWords) {
35       System.out.println("Checking for " + s3 + " " + myTrie.hasEntry(s3));
36     }
37   }
38 }
```

Listing 4: Console output after running `Main` Class

```
-------- Adding Words to Trie --------
Adding report true
Adding reporter true
Adding reporters true
Adding reported true
Adding reportable true
Adding reportage true
Adding reportages true
Adding reporting true
-------- Checking if all good words return true --------
Checking for report true
Checking for reporter true
Checking for reporters true
Checking for reported true
Checking for reportable true
Checking for reportage true
Checking for reportages true
Checking for reporting true
-------- Checking that all list of bad words return false --------
Checking for repo false
Checking for save false
Checking for t false
Checking for ejis false
Checking for repe false
```