Properties of Tries

(Normal / R) Tries

1. A trie is a special kind of rooted tree used to store strings composed over some alphabet \( \Sigma \). When a string such as \textit{algorithms} is input into the trie, a path from the root is constructed corresponding to the sequence of characters. Once the path \( a \rightarrow l \rightarrow g \cdots \) has been constructed, a special node is constructed with character $ to denote the end of a valid word.

2. The root of a trie represents an empty substring. Each node in a trie can have \(|\Sigma|\) children. Usually, these are stored in an array, as characters have a direct mapping to integers. Further, the descendants of a node \( v \) are all inserted words reachable on a path from \( v \) towards its children.

3. Tries implement basic functionality \texttt{insert(x)}, \texttt{find(x)}, and \texttt{delete(x)}. Time complexity of Trie insertion is usually defined over \( m \), the total string size of all inserted strings. With the algorithms we’ve discussed in class, trie construction runs in \( O(m) \) time. \texttt{find(x)} and \texttt{delete(x)} both run in \( O(l) \) time, where \( l \) is the length of the string being inserted or deleted.

Patricia Tries

Also known as compact tries, these data structures offer space usage improvements over simple tries. As we’ve discussed, a trie’s space usage is \( O(m) \) where \( m \) is the total size of all strings input. What property in a sequence of insertions fulfills this upper bound?

However, a simple motivation leads to an optimization. If a sequence of nodes all have one children, why have multiple nodes? In a compact trie, all such sequences are compacted into one node. A compact trie has \( O(s) \) nodes, where \( s \) is the total number of strings inserted into the dictionary. This can be proven as follows:

Upon insertion of a string, how many nodes are created? As you insert, you walk down the existing structure of the trie until you “fall off”, meaning no prior insertion of a string has had the same prefix. At this point, a new node is created, and this node stores the entirety of the rest of the string to be inserted. Insertion finishes, and no further nodes are created.

A final optimization is as follows: instead of storing strings at the nodes of our trie, we store every inserted string exactly once in an auxiliary array. Nodes then represent their character sequences by storing indices into the strings. Thus, if \textit{algorithms} is the first string I insert into my trie, I’ll have exactly 1 non-root node, with the data \((1, 1, 10)\). The first 1 points to the first entry in the auxiliary array. The 1, 10 mean that the node stores the substring from index 1 to index 10 (i.e. the whole string, in this case.)

Suffix Trees

A suffix tree is a trie whose stored strings are all suffixes of one long input sting. For the input string /textit{ENIAC}, a suffix trie holds

\[ \{C\}, \{A \rightarrow C\}, \{I \rightarrow A \rightarrow C\}, \{N \rightarrow I \rightarrow A \rightarrow C\}, \{E \rightarrow N \rightarrow I \rightarrow A \rightarrow C\} \]

Suffix trees are often used to search for substrings in a given string. You’ve probably heard the phrase “every substring is a prefix of some suffix.” This can be a little hard to grok at first. If a string is a sequence of characters \( S[1, \ldots m] \), then any substring is defined by some start index greater than or equal to 1, and some end index less than or equal to \( m \). We have \( m \) suffixes in our suffix tree, each one corresponding to a different start index. How many nodes we travel down the path of whichever suffix we’ve chosen determines the end index. Thus, by traversing down a portion of some suffix, we’re reading “a prefix of the suffix.”
A traditional suffix trie uses $O(m^2)$ space (consider storing a sequence of all unique characters.) A compact suffix trie uses just $O(m)$ space. The same space usage argument given for traditional tries, above, suffices for this.

**Ternary Search Tries**

We can see a clear problem with all the tries described above. Every node must have a child for every letter in the alphabet. This can lead to a huge space constraint absolute terms. Ternary Search Tries (TSTs) provide a way to cut down on this space problem by limiting the number of children a node has to 3.

We view a TST as consisting of layers where each layer represents a character in the string. A node in a TST has 3 children, left, right, and center. The left child of a node represents a character less that the value of the node which is in the same string position as the given node. These nodes are considered in the same layer. The right child of a node represents a character greater than the given node, again at the same string position. Finally, the center child represents a character following the given node on the next layer and in the next string position.

The root node in a TST is not empty but is instead a character in the first level. Similar to the tries above, a node has a non-null value if it is the last character of a string stored in the data structure. This value is the value of the string. The space of a TST is $O(m)$.

We will walk through the process of $\text{find}(x)$ in a TST and it’s time complexity. Beginning at the root of the tree and the first letter in the string $x$, we conduct the following procedure. If the node were currently at has a character value larger than the character were looking at in our sting we move to the left child. If the node were currently at has a character value smaller than the character were looking at in our sting we move to the right child. If the node were currently is the character in our string we move to the center child and move to the next letter in our string, $x$. If we ever reach a null node, we know $x$ is not in the TST and we return null. If we reach the end of $x$ we then return to value for the node were at. This will return the value associated with the string $x$ if it exists or null if the string does not exist.

In the worst case, we will need to traverse through $k$ nodes for each letter in our string $x$ where $k$ is the number of letters in our alphabet. Thus the time complexity of find is $O(k|x|)$

**Writing an Autocompletion Program**

**Formalizing the Problem**

As you type characters into your smartphone, its keyboard program predicts the word you’re writing. It does this by keeping track of the characters you’ve already typed in the current word, and making a guess at what words are most likely to complete the sequence so far.

Let alphabet $\Sigma$ be an ordered set of characters. Let $M$ be a vocabulary (a set of words) over $\Sigma$, with $M >> \Sigma$. Let $S$ be an unbounded sequence $(s_1, s_2, \ldots)$ of characters, with each $s_i \in \Sigma \cup \{\text{space}\}$ (Words can’t have a space character, but the sequence can.). By unbounded, we mean you receive the elements of $S$ in order, but don’t know any information past the last $s_i$ that you’ve received so far. Let the maximum number of words to present to the user at any time be $k$. This is usually bounded by screen size, and utility to the user. (If you’ve typed in nothing, do you want the whole English vocabulary given as a helpful suggestion?)

*Goal:* In a few steps, construct a data structure that permits the efficient and accurate prediction of words as sequence $S$ is received. Once there are $k$ or fewer possible word completions, keep track of the top $k$ as each character $s_i$ is received.

**Step 1: Adding Size Information to Tries**

Let’s say you have $M$ stored in a text file, one word per line, like a really bare dictionary. Modify trie insertion to keep track of an extra bit of information: for each node $v$, let size be the number of unique words that are children of $v$. What do you do upon insertion? What do you do upon deletion?

*Solution.* Upon insertion, at each node traversed, add 1 to size. When nodes are created, initialize their size to 1. Upon deletion, at each node traversed, subtract 1 from size.
Step 2: Extending to probabilities

The size field for each node is helpful in determining when there are $k$ or fewer predictions left. However, how do we know which of the $k$ words are the most likely completions? This kind of information is usually estimated by reading a large corpus, or body of text. Our method will be to read a corpus, and store the number of times each word is seen with its node in the trie. Thus, consider a corpus to be a sequence of insertion operations. Modify trie insertion again, this time keeping track of both size, and frequency, the number of times each word has been inserted into the trie. How can we make sure that we don’t increase the size field on nodes when inserting non-distinct words?

**Solution.** To ensure validity of size: When inserting a word, first check membership of the word in the trie. If the word is already a member of the trie, don’t modify size as it is inserted in the tree. To maintain frequency information, regardless of uniqueness of the word, add one to the frequency value for just the leaf representing the word when you reach it in insertion.

Step 3: Putting it all together

We now have a trie that stores the words in our vocabulary, as well as information at each node about the number of unique words “beneath” it. Further, we have frequency information in the form of integers associated with each leaf node (unique word). We now have the information we need to do probabilistic autocompletion. We now consider sequence $S$. Recall that this is the sequence of characters typed in by the user. Every time a character $s_i$ is keyed in, we receive that character, and decide what to do. Design a function $onChar(c)$ that takes character $c$ and returns a list $L$ of potential autocompletions with the following properties:

1. The sequence is size $k$ or smaller.
2. The sequence is ordered – that is, the first element is the most-likely word for autocompletion. Note that we’ll be estimating the “probability” as the frequency of each word divided by the total frequency of all possible words given the sequence so far.
3. $onChar(c)$ runs in $O(k \log(k))$ time.

Note that as we get more information (as we receive more characters in $S$,) our set of candidate completions become more accurate predictions. This is because we are effectively conditioning our probability calculations on the sequence so far, and more information → better predictions.

**Solution.** Initialize current, a pointer to the root node of the trie. As each $s_i$ is received, advance the pointer to the corresponding character in the trie. If $s_i$ would advance the pointer to a position not on the trie (a search miss), reset it to the root node, and keep it there until a “space” character is received. (Also, potentially underline the partial word in red? If our vocabulary is correct, there are no real words starting with the sequence so far.)

Once the pointer has been advanced, check the value of size at that node. If it is greater than $k$, return. Otherwise, run DFS/BFS from the current node. (What if we wanted to return the possible words in lexicographical order instead of probabilistic order? Talk about how we could do this with BFS!) Put the leaves that the graph traversal algorithm in $L$, our list of potential completions. Sort them by decreasing frequency value, and return them to the user.

Once we’ve constructed $L$ once, we don’t want to have to take $k \log k$ time to reconstruct it until we start a new word. Instead, once $L$ is constructed, upon the next $s_j$, before advancing the pointer according to $s_j$, DFS down all other children of the current node, and remove all leaves from $L$. This takes $O(k)$ instead of $O(k \log k)$.

Upon any $s_i = \text{space}$, reset the pointer to the root, and empty $L$ if it has any contents.

Testing your Understanding

**Problem 1.** Given some arbitrarily long string, how can we find the longest repeated substring? What about if we want the longest that is repeated $k$ times?
Solution. First we can build a suffix trie from the string as we process it. At any particular node we can check the number of internal nodes (which are valid words). Note that the deepest node with at least one internal node, meaning that the substring is repeated at least twice, will have the largest length (since depth tracks the number of characters in the word). Thus we can find the deepest node with at least one internal node and return that. For finding the longest substring repeated at least \( k \) times, we can do the same thing except now just keep track of nodes with at least \( k - 1 \) internal nodes and return the one with the largest depth.

Problem 2. Given a set of \( N \) strings, how can we find the longest common prefix between any two strings? Analyze the running time of your algorithm and give a short proof of correctness.

Solution. Initialize an integer to 0 that will keep track of the longest prefix seen so far, and initialize a string to \( \epsilon \) to keep track of the return value.

Add all of the strings into a trie, but modify the construction. Keep track, while inserting each string, of the current depth in the trie as long as no node has yet needed to be initialized. (Meaning some other string has already initialized this path upon its insertion.) If this depth becomes greater than the longest-prefix integer, store the substring for potential return.

Running time and correctness: This will take no more than (and hopefully much less than) \( O(m) \) time, where \( m \) is the total length of the strings inserted. (The construction time of a trie.) Correctness follows from the lemma that two strings \( w_1 \) and \( w_2 \) share a common prefix iff the insertion of the second, \( w_2 \), involves walking a path from the root of the trie to a node \( v \), such that the terminal node of \( w_1 \) is a descendant of \( v \). This, in turn, follows from there being a unique path from the root of a trie given any string \( w \), the length of the path being equal to the length of the string.