CS166 Problem Set 2: String Data Structures

Luis A. Perez

Problem One: Time/Space Tradeoffs in Tries (10 Points)

If you code up a trie, you'll quickly run into a design choice: how do you represent the child pointers associated with each node? This decision shouldn't be taken lightly; it will have an impact on the amount of time required to perform operations on the trie and on the space used to encode the trie. In this problem, you'll analyze the time/space tradeoffs involved in three different trie representations.

Here's some notation we'll use throughout this problem:

- We'll have k denote the total number of words stored in our trie.
- We'll have t denote the total number of nodes in the trie.
- We'll have Σ denote the alphabet from which the strings in our trie are drawn.

An assumption that's common throughout Theoryland that's also usually true in practice is that, given some character $x \in \Sigma$, it's possible to map x to some integer in $\{0, 1, 2, \ldots, |\Sigma| - 1\}$ in time $O(1)$, making it possible to do index-based lookups on characters efficiently. We'll also assume that each character fits comfortably into a machine word.

For the purposes of this problem, we'll imagine that each node in the trie is represented with some sort of structure that looks like this:

```
struct TrieNode {
    bool isWord;
    Collection<char, TrieNode*> children;
};
```

First, suppose you decide to store the child pointers in each trie node as an array of size $|\Sigma|$. Each pointer in that array points to the child labeled with that character and is null if there is no child with that label.

i. Give a big-Θ expression for the total memory usage of this trie. Briefly justify your answer.

> **Solution:** The total memory usage is $\Theta(t|\Sigma|)$. The trie has t nodes, and each node contains an array of pointers (word-sized) of size $|\Sigma|$ (plus an additional word for the boolean). Note that that this is a tight lower and upper bound, since the array is allocated for every node (even if there are no children).

ii. Give the big-O runtime of determining whether a string of length L in contained in a trie when the trie's children are represented as an array as in part (i). Briefly justify your answer.

> **Solution:** The runtime of termining wheter a string of length L is in the trie is $O(L)$. The string itself provides the indeces of each child array (in order) of which we must walk them. The string is in the trie if and only if we can walk the trie using all its characters (no NULLs encountered) and the final node we reach is marked as a word.

Now, let's imagine that you decide to store the child pointers in each trie node in a binary search tree. (A trie represented this way is sometimes called a ***ternary search tree***.)

iii. Give a big-Θ expression for the space used by a ternary search tree. As a way of checking your work, your overall expression should have no dependence on $|\Sigma|$. (Whoa! That's unexpected!) Briefly justify your answer.

> **Solution:** The total memory usage is $\Theta(t)$. The easiest way to explain is to break this into parts. First, our ternary search tree has t nodes, each which with one boolean. So ignoring the child-pointers, the nodes alone take $\Theta(t)$ space. Now, we note that each element in our BSTs consists of a single character and a single pointer, for space $\Theta(1)$. How many BST elements are there in the entire tree? Not any specific node, but overall? The answer is actually quite simple, if we notice that there's a one-to-one correspondance between elements and *TrieNode*s (ignoring the root node). Each pointer in a BST element points to a unique TrieNode (otherwise we wouldn't have a ternary tree), and each TrieNode is pointed to by a distinct pointer (the one in it's parent TrieNode) (except for the root node). As such, all of the BSTs added together take $\Theta(t)$ space.

iv. Suppose each node in the ternary search tree stores its children in a balanced BST. Briefly explain why the cost of a lookup in the ternary search tree is $O(L \log |\Sigma|)$.

> **Solution:** For each character in the word we're looking up, we need to perform a search for that character to know the next TrieNode. This search in a balanced BST takes $O(\log |\Sigma|)$, since in the worst case, all of possible characters in our alphabet exist as children of the TrieNode. We repeat this lookup $O(L)$-times, to give a total lookup time of $O(L \log |\Sigma|)$.

As you can see here, there's a time/space tradeoff involved in using ternary search trees as opposed to an array-based trie. The memory usage is lower for the ternary search tree than for the array-based trie, but the lookups are slightly slower.

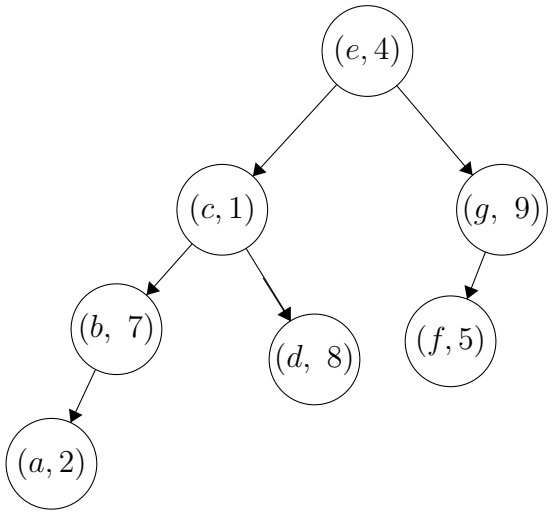
If you're working in computational biology where Σ is often {`A, C, T, G`}, then the cost of an extra factor of $|\Sigma|$ or $\log|\Sigma|$ isn't going to be too big. On the other hand, if you're working with general text, where Σ might be all of Unicode, including emojis, the impact of a $|\Sigma|$ or $\log|\Sigma|$ term can be enormous. Are there ways of representing tries where neither the time nor space depends on $|\Sigma|$?

Amazingly, the answer is yes, using a tool called a ***weight-balanced tree***. A weight-balanced tree is a binary search tree in which each key x_i has an associated weight $w_i > 0$. A weight-balanced tree is then defined as follows: the root node is chosen so that the difference in weights between the left and right subtrees is as close to zero as possible, and the left and right subtrees are then recursively constructed using the same rule.

v. Draw a weight-balanced tree containing the following key/weight pairs.

$$(a, 2) \quad (b, 7) \quad (c, 1) \quad (d, 8) \quad (e, 4) \quad (f, 5) \quad (g, 9)$$

Solution:



vi. Suppose that the total weight in a weight-balanced tree is W. Prove that there is a universal constant ε (that is, a constant that doesn't depend on W or the specific tree chosen) where $0 < \varepsilon < 1$ and the left subtree and right subtree of a weight-balanced tree each have weight at most εW.

Solution: We show that our universal constant can be any value in $\frac{2}{3} < \varepsilon < 1$.

For contradiction, suppose there exists some weight balanced tree such that at least one child has weight $L \geq \varepsilon W$. WLOG, let us take this to be the left-substree. Then the weight of the right subtree is $R < (1 - \varepsilon)W$ (because $w_R > 0$). This implies that

the difference in weight between the two substrees is at least

$$\begin{aligned} D &= L - R && (L > R) \\ &> \varepsilon W - (1-\varepsilon)W \\ &\text{(Substitute smallest and largest values for L and R respectively)} \\ &= (2\varepsilon - 1)W \end{aligned}$$

. We also have that the weight of the root node is:

$$\begin{aligned} w_R &= W - (L + R) && \text{(definition)} \\ &\leq W - L && \text{(upper bound if right side is empty)} \\ &\leq (1-\varepsilon)W && \text{(smallest possible value of L substituted)} \end{aligned}$$

However, we now construct a new tree with a smaller difference, contradicting the fact that the above tree is weight balanced.

Simply take the right-most node in the left-substree (which must exist, since it's weight is not zero), with weight $w_L > 0$, and make this the new root. Similarly, take the root and move it into the right-substree.

We now note that the weight of the new left subtree is

$$\begin{aligned} L' &= (L - w_L) && \text{(removed a node with } w_L \text{ weight)} \\ &< L && (w_L > 0) \end{aligned}$$

and the right subtree is

$$\begin{aligned} R' &= (R + w_R) && \text{(added root node)} \\ &> R && (w_R > 0) \end{aligned}$$

We now have two cases to consider.

For the first case, suppose $L' \geq R'$ (still), then this implies that the new difference in weight is:

$$\begin{aligned} D' &= L' - R' \\ &= (L - w_L) - (R - w_R) \\ &= (L - R) - (w_R + w_L) \\ &< (L - R) && (w_R + w_L > 0) \end{aligned}$$

Therefore this new tree was a smaller difference, contradicting the assumption that the original tree is weight balanced.

For the second case, suppose $L' < R'$ (left-tree became smaller). Then the new

difference in weight is:

$$\begin{aligned} D' &= R' - L' \\ &= (R + w_R) - (L - w_L) \\ &= (R - L) + (w_R + w_L) \\ &= W - 2L + w_L && \text{(Substitute } w_R = W - (L + R)\text{)} \\ &\leq W - L && (w_L \leq L \text{ since it came from the left subtree}) \\ &\leq (1 - \varepsilon)W && (L \geq \varepsilon W \text{ by our assumptions}) \end{aligned}$$

. What we want, at the end of the day, is an ε such that $D' < D$. Substituting our work from above, we're looking for a value of ε such that:

$$D' \leq (1 - \varepsilon)W < D = (2\varepsilon - 1)W$$

.
Solving this inequality, we show that this holds for all $\varepsilon > \frac{2}{3}$.
This concludes our proof. We can pick any $\varepsilon \in (\frac{2}{3}, 1)$ as our universal constant, and this value does not depend on W or the specific tree.

Let's now bring things back around to tries. Imagine that we start with a ternary search tree, which already stores child pointers in a BST, but instead of using a regular balanced BST, we instead store the child pointers for each node in weight-balanced trees where the weight associated with each BST node is the number of strings stored in the subtrie associated with the given child pointer.

For example, consider the trie from Slide 10 of our lecture on suffix trees. Focus on the node associated with the word `ant`. Then its child labeled `e` would have weight 3, since that subtrie contains three words, and its child labeled `i` would have weight 1, since that subtrie contains just one word.

vii. Prove that looking up a string of length L in a trie represented this way takes time $O(L + \log k)$. As a hint, imagine that you have two bank accounts, one with $O(L)$ dollars in it and one with $O(\log k)$ dollars with it. Explain how to charge each unit of work done by a search in a trie represented this way to one of the two accounts without ever overdrawing your funds.

Solution: As per the hint, consider that we have two bank accounts, A and B. Now let us classify pointers in our trie structure. Following pointers to *TrieNode*s is one unit of work, and this will be charged to A. Following pointers to elements in our weighed BST are also one unit of work, and this will be charged to account B. Then note that during the search we follow $O(L)$ trie pointers, since there are L characters in our input string and each character will correspond to one TrieNode. As such, we will never overdraft our A bank account.

Now, to show that we never overdraft our B account, we have to be a little more clever. We note that by our results from problem (vi), each time we follow a weighed BST pointer, the weight of our the new child we land on is at most ϵW where $0 < \epsilon < 1$. Since the weight of the BST corresponds to the number of strings in the subtrie associated with it, this means that each time we follow a weight BST pointer, we decrease the number of strings we're searching by a factor of ϵ. Therefore, this can occur at most $O(\log k)$ times (since at this point, the subtrie has only one string associated with it).

Put together, the above gives the overall runtime which consists of following *TrieNode* pointers plus following weighed BST pointers as $O(L + \log k)$, as desired.

Weight-balanced trees have a bunch of other wonderful properties, and we'll spin back around to them later in the quarter.

There are a bunch of other ways you can encode tries, each of which has its advantages and disadvantages. If you liked this problem, consider looking into this as a topic for your final project!

Problem Two: The Anatomy of Suffix Trees (2 Points)

Consider the following paragraph, which is taken from an English translation of the excellent short story "Before the Law" by Franz Kafka:

```
Before the law sits a gatekeeper.  To this gatekeeper comes a man from
the country who asks to gain entry into the law.  But the gatekeeper
says that he cannot grant him entry at the moment.  The man thinks about
it and then asks if he will be allowed to come in later on.  "It is
possible," says the gatekeeper, "but not now."
```

Actually building a suffix tree for this text, by hand, would be quite challenging. But even without doing so, you can still infer much about what it would look like.

i. Without building a suffix tree or suffix array for the above text, determine whether the suffix tree for this passage contains a node labeled with the string `moment`. Briefly justify your answer.

> **Solution:** The suffix tree does not contain a node labeled with the string `moment`. All leaf-node labels in suffix trees are suffixes, of which `moment` is not. Furthermore, `moment` is not a branching word (it doesn't even repeat), and as such, it is not the label of any internal nodes.

ii. Repeat the above exercise for the string `man`.

> **Solution:** The suffix tree does not contain a node labeled with the string `man`. Again, this string is not a suffix (and therefore not a leaf-label). Furthermore, it is not a branching word since in both occurrences, it is a followed by a space. Therefore, it is also not the label of any internal nodes.

iii. Repeat the above exercise for the string `gatekeeper`.

> **Solution:** The suffix tree indeed does contain a node labeled with the string `gatekeeper`. This is because `gatekeeper` is a branching word, since either a space ( ) or a period (.) can be appended and both occur in the text. Therefore, we have an internal node with the label `gatekeeper`.

Problem Three: Longest k-Repeated Substrings (3 Points)

Design an O(m)-time algorithm that, given a string T of length m and a positive integer k, returns the longest substring that appears in T in at least k different places. The substrings are allowed to overlap with one another; for example, given the string `HAHAHAHAHA` and $k = 3$, you'd return `HAHAHA`.

As a note, the runtime of O(m) is independent of k. For full credit, your solution use should not use suffix trees, though you're welcome to use any other data structures you'd like.

Solution: Overview: The key insight is that we've already solved this problem in $O(m)$ for $k = 2$ with an LCP-array. The next insight is that by taking the minimum over w adjacent elements in the LCP array, we actually find the the longest substring shared by the i-th, $(i + 1)$-th, ..., $(i + w)$ suffixes. Since these suffixes are in sorted order, if we just take the max over these minimums for all i and return the string corresponding to it, we will have found the longest substring that appears in T in at least k different places.

Algorithm Detail: When given as input T, we first construct a rank-order representation of T (similar to our programming assignments). Note that this requires us to assume that Σ is constant and relatively small. We then use our SA-IS algorithm to construct the suffix array from T. We construct a reverse mapping from suffix-array values to suffix array indeces simply by constructing a new array and iterating over the suffix array, storing indeces in as values with the value as their index. With this reverse mapping, we can directly apply Kasai's algorithm (the reverse mapping allows us to find the suffix in the suffix array) and construct the LCP-array.

Once we have the LCP-array for T, we want to construct a new array containing the minimum over windows of size $k - 1$. Doing this in linear time is not trivial. We describe one possible algorithm for doing so now.

The sub-problem we wish to solve is finding the minimum of windows of size w in an array A. We allocate two new arrays $leftToRightMin$ and $rightToLeftMin$ of the same size as A. We then iterate from left-to-right over A, storing the smallest value seen so far into the corresponding index in $leftToRightMin$, with the additional modification that after seeing w values, we take the $w + 1$-th value as the min. We repeat this process from right-to-left to fill the $rightToLeft$ array.

An example will elucidate our description. Suppose we have $w = 3$ and $A = [4, 2, 5, 8, 7, 2, 10]$. Then we'll have $leftToRightMin = [4, 2, 2, \mid 8, 7, 2, \mid 10]$ and $rightToLeft = [4, \mid 2, 2, 8 \mid 2, 2, 10]$.

Now, to compute the minimum of the window $[i, i + w - 1]$ (inclusive), we simply compute the minimum of $leftToRight[i + w - 1]$ and $rightToLeft[i]$. Now back to our originally scheduled programming. For illustrative purposes we dropped the indexes, but note that we can keep track of them by attaching the to their values and ignoring them.

Once we have all of the minimums for windows (along with their indexes in the LCP-array), we find the maximum value of these minimums, along with it's corresponding index in the

LCP-Array. We the use this index to find the suffix offset in the suffix array, and return the corresponding string (using the maximum we calculated to know how many characters to return), all in constant time.

Correctness: The proof of correctness is also rather straight-forward. Given the LCP-array, H, we know that $H[i]$ corresponds to the length of the LCP of the i-th and $(i+1)$-th suffixes. This implies that the minimum over a window of size w corresponds to the length of the LCP of the i-th, $(i+1)$-th, ..., $(i+w)$-th suffixes (taking the minimum means these characters must match across all these suffixes). Therefore, the array we constructed of minimums of windows of size $k-1$ contains the lenghts of the longest common prefix across k suffixes. Since we then return the maximum and the suffixes are in sorted order, we must have returned the longest substring that appears in T at least k times.

Run-Time Complexity: The run-time is straight forward. Processing and building the suffix array and LCP array are all $O(m)$. Constructing the minimum of windows is also $O(m)$, since it takes just three passes over the LCP-array. Finding the maximum over our minimums is also $O(m)$, and finally returning the appropriate string takes constant time (just a few indexing operations).

As such, the total run-time complexity of the algorithm is $O(m)$.

Problem Three: Suffix Array Search (3 Points)

This one is all coding! Make sure to submit your implementation on `myth`.

Problem Four: Implementing SA-IS (12 Points)

This is one is also all coding! Make sure to submit your implementation on `myth`.