Text Indexing with Wildcards: An Experimental Comparison

6.851 Final Project

Hayden Metsky (hmetsky@mit.edu) and Casey O'Brien (cmobrien@mit.edu)

May 18, 2014

1 Introduction

Consider the text indexing problem. Given an input text and query string, we wish to report all the indices of the input text to which the query string matches. In this paper we will consider a generalized version of this problem, in which we allow the query string to contain *wildcards*. A wildcard is a character which can be matched to any other character. We will discuss four possible approaches to this problem. The first two are naive solutions, one of which uses little space but has a slow query time, and the other which has a fast query time but uses a lot of space. Then, we will discuss two approaches presented by Cole et. al in 2004 [1].

While theoretical bounds for each of these solutions have already been established, our goal is to report the performance of each of these solutions in practice. To this end, we implemented each of them in Java. For each solution, we measured the total space and query time for various texts and query patterns, allowing varying numbers of wildcards.

In terms of space, the solutions performed as expected based on their theoretical bounds. Perhaps more interestingly, the actual performances of the query algorithms differed substantially from what was suggested by their theoretical runtimes. In particular, the asymptotically better solution by Cole et. al performed worse than both naive approaches.

1.1 Organization of the Report

In section 2 we will present an overview of each of the solutions as well as their theoretical bounds. In Section 3, we will discuss the details of the implementation of our structures and the methods we used to measure their performance. In Section 4, we present the results of our experimentation, and finally in section 5 we discuss the underlying reasons for and implications of these results.

1.2 Notation

Throughout this paper, for ease of notation, we will interchangeably use t to refer to the query text itself and also the length of the query text (and similarly for p). We will use k to denote the maximum number of wildcards allowed in a query. Note that because the building of the structures can depend on k, we require that this parameter be set beforehand and not dynamically with each query. Finally, we will use Σ to denote the size of the alphabet.

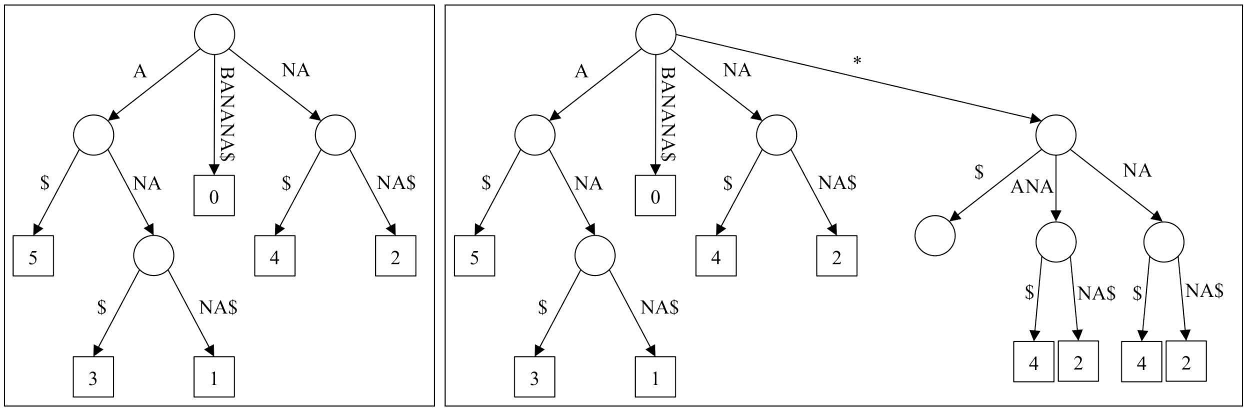

Figure 1: On the left is the standard suffix tree for t = BANANA. On the right, the suffix tree has been modified to contain a wildcard subtree at the root. Note that the big space, fast query solution requires wildcard subtrees be created between every letter.

2 Survey of Solutions to the Text Indexing Problem

2.1 Naive Solution 1: Small Space, Slow Query

This solution uses only a standard suffix tree S on t. To perform the query, we perform a standard query on the suffix tree, with the addition that each time we come across a wildcard in p, we branch and explore all possible subtrees. The space for this solution is clearly $O(t)$, and the time for a query is $O(\Sigma^k p)$. The Σ^k term arises from the fact that we may have to fully explore every subtree of a node each time we reach a wildcard. We will refer to this solution as the Slow-Query approach.

2.2 Naive Solution 2: Big Space, Fast Query

By modifying the suffix tree to store additional information, we can achieve a much faster query time. For this solution, we modify each node v to contain an edge to a wildcard subtree. This edge will represent the wildcard. We build a suffix tree from all the suffixes represented below v, skipping over the first letter of each suffix. We build these wildcard subtrees recursively, such that each root-to-leaf path contains at most k wildcards. Note that building these subtrees will involve splitting edges, because we need to have the option to match a wildcard instead of any given letter. Figure 1.1 shows an example suffix tree with a single wildcard tree built. With these wildcard subtrees, we have a query time of $O(p)$. However, the space required to store this tree is $O(t^{1+k})$. We will refer to this solution as the Big-Space approach.

2.3 Cole et al. Solution 1: Centroid Path Decomposition with Slow Query

In 2004, Cole et al. provided a solution (that we call CPD-Slow) to this problem which achieves faster query time that the first solution, but less space than the second solution. In order to achieve this, the suffix tree is first decomposed into *centroid paths*. The centroid path from the root to one of its descendant leaves is the path which follows the edges with the most descendant leaves (breaking ties arbitrarily). For nodes hanging off of this centroid path, centroid paths are created

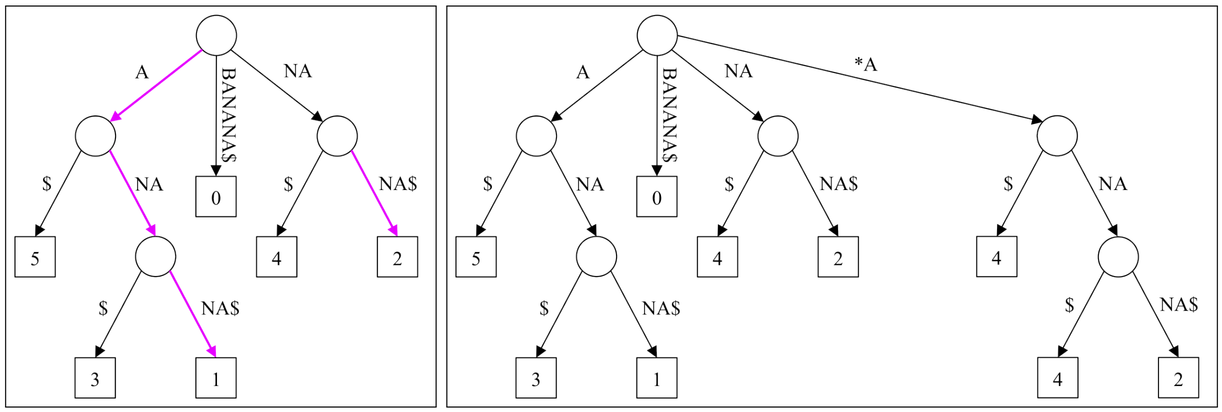

Figure 2: On the left is the standard suffix tree for t = BANANA, with centroid edges highlighted. On the right, the suffix tree has been modified to contain a CPD wildcard subtree at the root.

recursively. The centroid edge of a node is the outgoing edge of the node which lies along a centroid path. By construction, each node has exactly one centroid edge. Figure 1.2 shows the centroid path decomposition of an example suffix tree.

Like the big space solution described above, the Centroid Path Decomposition (CPD) solution builds wildcard subtrees. However, the wildcard subtrees only include suffixes below the node which do not lie along the centroid edge of that node. With this construction, the space to store the suffix tree is only $O(t \log^k t)$.

The general idea for the query is that whenever we reach a wildcard, we have to explore both the edge along the centroid path and the wildcard edge. Formally, to perform a query, we begin by dividing the query $p = p_0 * p_1 * \ldots * p_k$, where each of the segments p_i contain no wildcards. We begin by performing the following steps for $i = 0$, and continue until we have either finished with $i = k$ or have discovered a place where p diverges from t.

1. Perform a standard query for p_i. Say that this query ends at position v in the tree (which could be a node or along an edge).

2. If v has depth equal to the length of p_i below the starting point, then the query continues. Otherwise p_i must diverge from t and we return no matches.

3. If v is a node:

   (a) Advance once character along the centroid path (to account for the wildcard), and then recurse from that location on p_{i+1}.

   (b) Recurse from the root of the wildcard subtree rooted at v on p_{i+1}.

4. Otherwise v is a point along an edge. Advance one character along the edge and then recurse from that location on p_{i+1}.

If the query returned locations within the tree, we report all the leaf descendants of those locations as matches. For each wildcard, we might branch and explore two different edges. Thus, the total time for this query is $O(2^k p)$.

2.4 Cole et al. Solution 2: Centroid Path Decomposition with Fast Query

In their paper, Cole et. al suggested a way (that we called CPD-Fast) to speed up the query time by using a carefully constructed longest common prefix (LCP) query instead of walking down the tree as we did in Step (1) above. The structure is the same as that described in the previous section. We will need to augment the tree with additional information, but we will be sure that the information never takes more than linear space in the size of the tree. Thus the space is still $O(t \log^k t)$.

This new algorithms calls for preprocessing of p which takes time $O(p)$. The LCP query takes time $O(\log \log t)$. When we replace Step (1) from above with this LCP query, executing the entire query will take time $O(2^k \log \log t)$. Thus, the overall time of the query will be $O(p + 2^k \log \log t)$.

Let S be the original suffix tree on t. In order to be able to answer LCP queries, we need to build some structures into our suffix tree. We will refer to these structures collectively as the LCP structure. An LCP structure needs to be built for each subtree hanging off a centroid path. All LCP structures are built as part of the preprocessing of the text.

Next we describe the LCP query. Formally, the LCP query takes a node r and query text p_i (with no wildcards), and returns the location v which corresponds to the deepest location at which p_i could be matched, starting at r. For now, we will assume that the subtree rooted at r contains its own LCP structure. Because we don't build an LCP structure on every subtree, this may not actually be the case. After we describe the query, we will describe how to handle the case where r is not at the root of some LCP structure.

Let T be an arbitrary subtree on which we are building the LCP structure. First, the LCP structure on T needs to be able to return the LCA (least common ancestor) of any two nodes in T. This can be done in space $O(|T|)$ with constant time queries using the techniques described in [2].

Next, the LCP structure needs to support weighted level ancestor (WLA) queries. Given a node v and a height h, a WLA query should return the position at height h above v in T. The (unweighted) level ancestor problem can be solved in constant time and $O(|T|)$ space [2]. The same ideas can be used to solve the weighted level ancestor problem, but the query time increases to $O(\log \log t)$ due to the need to make a predecessor query.

Finally, each leaf in T needs to store the index of its corresponding suffix in lexicographic order in S. We will refer to this value as I_x, where l_x is some leaf node. Note that we store the index of this suffix in S, even though we are currently building a LCP structure on T. Then, given the index I_x of a leaf, we need to return the two leaves in T which have indices which are the predecessor and successor of I_x. This can be done in $O(\log \log t)$ by storing the indices of all the leaves of T in a y-fast trie.

Recall that we have divided p into $p_0 * p_1 * \ldots * p_k$. When we receive a query p, we perform the following preprocessing for each p_i. Our goal is to determine, for each p_i, the *index* g_i of p_i, and also the leaf with which p_i has the most overlap. We determine this value as follows.

1. Proceed like a standard query in S for p_i, noting the endpoint.
2. Compute the leaf l_i corresponding to the closest string to p_i, lexicographically. This can be found easily as long as each node has a pointer to its leftmost and rightmost leaf descendants. Also store h_i, the length of the longest common prefix of the suffix on l_i and p_i.

3. If the suffix at l_i comes before p_i lexicographically, $g_i = I_i + \frac{1}{2}$. Otherwise $g_i = I_i - \frac{1}{2}$.

It takes $O(p_i)$ to perform the preprocessing for any given p_i, so the total time to perform the preprocessing is $O(p)$. Once we have completed the preprocessing, then we are prepared to complete an LCP query. The query is computed as described below.

1. Look up the leaves in T with lexicographic indices corresponding to the predecessor (call this l_p) and successor (call this l_s) of g_i. Each leaf should also store a pointer to the leaf in S with the same index.
2. Compute h_p, the depth of the node $LCA(l_p, l_i)$ in S, and symmetrically h_s from $LCA(l_s, l_i)$.
3. Compute $\max\{h_p, h_s\}$. Without loss of generality assume the maximum was h_p. All further steps are symmetric if h_s is chosen.
4. Compute $h = \min\{h_i, h_p\}$. The represents the amount of overlap between p_i and T.
5. Let d_p be the depth of the predecessor. Return $WLA(l_p, d_p - h)$.

Steps (1) and (5) take $O(\log\log t)$ due to the structures described above, and Steps (2), (3), and (4) take constant time. Thus, the overall time to complete this query is $O(\log\log t)$.

Finally, we need to handle the case where we want to perform an LCP query on a node r which is not at the root of an LCP structure. To perform this query, we will carefully choose a descendant node which is the root of an LCP structure, and then perform the LCP query from there. Let T be the subtree which is rooted at the nearest ancestor of r with a LCP structure. We perform an LCP query on a node not at the root of an LCP structure as follows.

1. Let C be the centroid path in T which contains r (each node stores a pointer to its centroid path).
2. Find the leaf l_u in S corresponding to the suffix represented by the portion of C below r. To do so:
   (a) Get the offset in t of the leaf in T at the end of C (each leaf stores this information).
   (b) The offset of l_u in t is equal to this offset plus the depth of r in T.
   (c) Look up the leaf in S corresponding to this offset (we can store a table of these pointers).
3. Compute h' as the depth of node $LCA(l_u, l_i)$ in S.
4. Let $h = \min\{h_i, h'\}$.
5. Find the position v at height h below r on C. To do this, for each centroid path we store a y-fast trie containing depths of all the nodes along C.
6. If v is not a node, then return position v. Otherwise, perform another LCP query on the child of v corresponding to the next letter of p_i. This child will be the root of an LCP structure since it is hanging off a centroid path.

Steps (2), (5), and (6) take $O(\log\log t)$, and steps (1), (3), and (4) can be done in constant time. Thus, we have an overall LCP query time of $O(\log\log t)$ for any node.

Now we can perform a query by first preprocessing p, and then continuing with the steps outlined in Section 1.5, with the modification that instead of performing a standard query in Step(1) we perform an LCP query. The overall time of this algorithm is then $O(p + 2^k \log\log t)$.

3 Methodology

3.1 Implementation of the Approaches

We implemented the four approaches in the Java programming language. While other languages may yield better performance, our primary objective is to measure relative performance between the approaches rather than strive for an optimal absolute performance. Java is the language with which we are most comfortable, and it was therefore a natural choice.

We began by implementing a standard suffix tree to support the Slow-Query approach[1]. Indeed, this is the simplest data structure because the structure itself has no recursion beyond what is in a typical trie; furthermore, the query is a straightforward recursive algorithm. To construct the tree, we implemented Ukkonen's algorithm, which is an online algorithm that runs in $O(t)$ time. Our implementation of the algorithm was based heavily on a StackOverflow post [3] and on an existing implementation [4]. In fact, while we initially believed an online approach would be helpful for later components of the implementation, and thus chose this algorithm, it turned out to not be necessary; any linear-time algorithm for constructing a suffix tree would have sufficed. Prior to constructing the tree, we read the input text once and store it once. We built a collection of small structures[2] to easily work with substrings of the text, where each substring is simply stored using pointers to the original text. Thus, each edge can store a substring of the text with $O(1)$ space, as desired.

We then proceeded by generalizing the standard suffix tree to recursively build wildcard subtrees down to a depth of k wildcards[3]. This enables us to support the Big-Space approach. Like the Slow-Query approach, the queries in this approach are implemented in a relatively straightforward manner using recursion.

We modified the Big-Space data structure to support the CPD-Slow and CPD-Fast structures[4]. Namely, when we build a wildcard subtree off a node v, we copy the entire subtree of v except the part under its centroid edge. We then devised and implemented an algorithm (that we call `condense`) that recompresses this subtree in linear-time. One variant of this structure (CPD-Slow) simply stores these wildcard subtrees and supports queries as described in Section 2.3. It does not store any extra information with each wildcard subtree beyond what is necessary. The second variant of the structure (CPD-Fast) associates with each wildcard subtree information that allow us to perform efficient LCP queries as described in Section 2.4.

In particular, for this second variant we implemented from scratch, using 6.851 notes as reference,

[1]In our code, this is the `stringmatch.ds.suffixtree.SuffixTree` class.
[2]In our code, these are in the `stringmatch.ds.text` package.
[3]In our code, this is the `stringmatch.ds.suffixtree.SuffixTreeNaiveBigSpace` class.
[4]In our code, this is the `stringmatch.ds.suffixtree.SuffixTreeeWithCPD` class.

structures to support LCA queries and WLA queries, as well as a y-fast trie to support predecessor and successor queries on indices of a subtree. We store these structures in each wildcard subtree[5]. In implementing each of these structures we needed to support worst-case constant-time lookup in a hash table. To do this, we used a Java implementation of Cuckoo hashing found online [5].

3.2 Measuring Performance of the Approaches

3.2.1 Dataset and Trials

To measure and compare the perform of the data structures we implemented, we used the Open American National Corpus (OANC) [6]. This provided us with 14588048 words. We stripped this corpus of all whitespace and other non-alphabetic characters, and converted the remaining characters to uppercase, leaving us with an alphabet size of $\Sigma = 26$. After this preprocessing, our input text contains a total of 68576990 characters.

There are four parameters that can affect space usage and runtime: Σ, p, t, and k. We always leave Σ fixed and usually leave p fixed. For most runs we vary t and k, and the particular values we use are given in Section 4.

Using the OANC dataset, we generate random input to use in evaluation of space usage and runtime.[6] In particular, for some fixed collection of the four parameters, we run five "trials" on a data structure. In each trial, we select a random length t substring of the OANC corpus, and this substring becomes the input text to the data structure being evaluated. We build the data structure at each trial and (just on the first trial) determine and record its space usage. Then, for the current trial, we randomly select 2000 length p substrings from the input text and, in each of these substrings, replace k random characters with wildcards; these make up our queries. We time each query and report the average time across all 10000 queries as the average query time for the data structure being evaluated.

3.2.2 Instrumentation of Space and Time Resources

To measure the space usage of each data structure for varied parameters, we use the Jamm MemoryMeter [7], which acts as a wrapper around Java's Instrumentation package for measuring object size. Because of Java's handling of memory, measuring space usage is tricky and we found that the exact space usage of an identical experiment varied notably depending on the machine on which the experiment was being run. However, the tool should still provide a rough estimate for considering relative space usage because we are consistent in running the experiments (e.g., on the same machine).

We implement a measurement of average query time for a set of queries by computing the difference in the clock's time over the set of queries. However, we encountered substantial difficulty after initially implementing this approach and running experiments. In particular, the average query time would differ greatly across experiments and would create a nonsensical pattern as we varied a parameter (e.g., it would sometimes be very high for even values of k and low for odd values).

[5]The LCA and WLA structures can be found directly in the `stringmatch.ds.suffixtree.SuffixTreeeWithCPD` class. The y-fast trie can be found in the `stringmatch.ds.suffixtree.yfasttrie` package.

[6]In our code, this evaluation suite is in the `stringmatch.evaluate.Evaluator` class.

We eventually narrowed the cause of this problem to Java's memory management — namely, we determined that its garbage collection would substantially slow arbitrary queries. We experimented with some tricks, like calling `System.gc()` in between trials, to alleviate these issues, and they helped somewhat. However, we determined that one approach to better avoid this issue would be to give Java far more heap space than it would need in building the data structures on our chosen parameter values.

Consequently, we ran our experiments on a large memory instance from Amazon Web Services (AWS).[7] In doing so, we gave Java 200 GB of heap space. This led to smoother plots that, we believe, more accurately reflect the query times of the data structures. We monitored the process' memory usage to ensure it did not approach the heap limit.

(The figures and data in our May 14, 2014 presentation were generated from runs on one of our personal machines, which has far less available memory. The substantial increase in available memory offered by running experiments on AWS also allowed us to experiment with larger parameter values. For these reasons, the figures and data in this paper differ from what we showed in our presentation.)

4 Results

5 Discussion

References

[1] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 2004.

[2] Erik Demaine. Least Common and Level Ancestors. MIT 6.851 Lectures 15 Notes Spring 2014. http://courses.csail.mit.edu/6.851/spring14/lectures/L15.pdf

[3] StackOverflow. *Ukkonen's suffix tree algorithm in plain English?* March 2012 — May 2013. http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/

[4] Max Garfinkel, GitHub. *A java implementation of Ukkonen's suffix tree creation algorithm.* September 2012. https://github.com/maxgarfinkel/suffixTree/

[5] Keith Schwarz. *Cuckoo-Hashmap.* November 2010 — December 2013. https://github.com/maxgarfinkel/suffixTree/

[6] The Open American National Corpus. http://www.anc.org/data/oanc/

[7] Jonathan Ellis. *Java Agent for Memory Measurements.* February 2011 — May 2014. https://github.com/jbellis/jamm

[7]In particular, we used its `r3.8xlarge` instance, which offers 244 GB of memory.