**6.897: Advanced Data Structures** Spring 2005

# Lecture 19 — April 14, 2005

*Prof. Erik Demaine* *Scribe: Vincent Yeung*

## 1 Overview

In this lecture, we discuss the problem of approximate string matching. In particular, we outline solutions for solving the exact matching problem for patterns with "don't care" symbols, denoted by ?. In the process, we will be using solutions for the level ancestor problem, which we also discuss.

## 2 Approximate String Matching

The approximate string matching problem is defined as follows. Given an error tolerance $k$ and a text $T$, construct a data structure which can answer the following query: find occurrences of a pattern $P$ in $T$ within *error* $k$. There are different ways to measure error, such as:

1. Hamming distance: the number of character mismatches
2. edit distance: the number of edits (insertions, deletions, substitutions) needed to produce an exact match.

The best currently-known bounds, given by [4], are:

- space and preprocessing: $O(|T|\frac{(c\lg|T|)^k}{k!})$
- query: $O(|P| + \frac{(c\lg|T|)^k}{k!}\lg\lg|T|) + 3^k \cdot (\#\text{ occurrences})$

We will not actually cover this data structure, but we concentrate on a simpler problem where the same techniques are used. These bounds are only interesting for small $k$ (such as a constant). For larger $k$, there are relaxations of the problem which can be solved more efficiently. This will be the topic of next lecture.

## 3 Searching with Wildcards

We will focus on a subproblem of the above. Again we are given $T$ and $k$ for preprocessing. But now, the query consists of a pattern $P$ that contains at most $k$ "don't care" characters (the ? wildcards). We are to find "exact" matches of $P$, where wildcards match any character. The best known solution [4] solves the problem in $O(|T|\lg^k|T|)$ space and $O(2^k\lg\lg|T|+|P|+\#\text{ occurrences})$ query time.

All the solutions we will discuss involve the use of suffix trees. An obvious simple solution is to walk down the suffix tree while matching $P$ and simply branch $|\Sigma|$ ways every time a ? is encountered in $P$. Thus, queries take at most $O(|\Sigma|^k \cdot |P|)$. Compared to the best solution we mentioned above, the simple solution is lacking in that there is a dependence on alphabet size (which may be very large) and that the dependence on the pattern length is multiplicative instead of additive.

We now describe how to improve the $\Sigma^k$ factor to $2^k$. To do so, we perform a heavy-light decomposition of the suffix tree. Recall that an edge to a child is *light* if the subtree rooted in that child contains at most half the nodes of the parent's subtree. The intuition is that we now only differentiate between light edges and possibly a single heavy edge whenever we encounter a ?. Because light subtrees are small, we group them together in one big chunk. Specifically, for each node in the *primary* suffix tree, we store a *secondary* suffix tree on the union of light subtrees of that node, except the first characters of each subtree.

If $k > 1$, we recurse $k$ times so that there are $k + 1$ "levels" of secondary trees. Since the light depth is $O(\lg |T|)$ in a heavy-light decomposition, each leaf appears in $O(\lg^k |T|)$ trees. Thus, the solution takes $O(|T| \lg^k |T|)$ space and preprocessing, and $O(2^k \cdot |P|)$ query time.

As mentioned above, there is a way to make the $|P|$ factor additive in the query time. The idea is to find a way to quickly (in $\lg \lg |T|$ time) determine whether we should take the light/heavy branch. We will not delve into the specifics of this solution, but mentioned that using the suffix tree from above, least common ancestor queries, and *level ancestor* queries, we can detect whether one of the $2^k$ branches is "good".

# 4 The Level Ancestor Problem

For the rest of this lecture, we shift our attention to the level ancestor problem. We are given a static rooted tree, which can be preprocessed. Then, a level-ancestor query is given a node $V$ and a number $l$, and must find the $l^{\text{th}}$ ancestor of $V$. This is equivalent to finding the depth-$d$ ancestor of $v$, where $d + l = \text{depth}(V)$.

Various solutions to this problem have been proposed [3, 5, 1, 2]. We will discuss the solution in [2], by Bender and Farach-Colton. We present gradual steps leading to a solution that encompasses the different improvements, and ends up taking linear space and preprocessing time, with constant query time. First observe that an immediate solution is to store a lookup table for each node. This gives total space $O(n^2)$ and constant query time.

**Jump pointers.** Think of skip lists. With *jump pointers*, each node stores its 1st, 2nd, 4th, ..., $2^i$-th ancestors. This takes $O(n \lg n)$ space. To perform queries, recursively go up $\lfloor \lfloor l \rfloor \rfloor = 2^{\lfloor \lg l \rfloor}$. We know that $l/2 < \lfloor \lfloor l \rfloor \rfloor \leq l$, so queries take $O(\lg n)$.

**Long path decomposition.** We preprocess the tree as follows:

1. take a longest root-to-leaf path and recurse on the remaining connected components.
2. store each path as an array ordered by depth (so that nodes in the path may be randomly accessed), and store a pointer to its parent path.
3. for each node, store the path to which it belongs and its index in the array for the path.

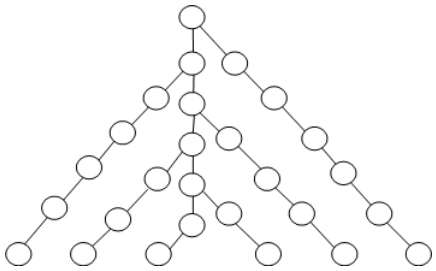

Figure 1: The maximum path depth in the long path decomposition can be as high as $O(\sqrt{n})$.

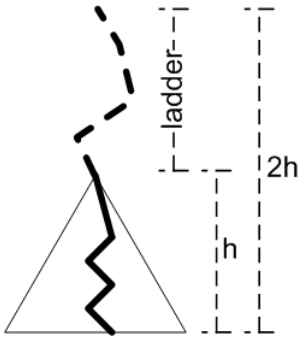


Figure 2: The ladder decomposition.

Let HEIGHT($v$) be the height of node $v$ in its path, i.e. the number of nodes beneath it.

The space usage is clearly $O(n)$. To answer queries, get the path for the node and check if it is long enough for the queried ancestor height; if not, recurse. The query time is therefore linear in the number of paths traversed. Unfortunately, Figure 1 shows that the number of paths can be as high as $O(\sqrt{n})$.

**Ladder decomposition.** This extends the long path decomposition in a simple but effective fashion. We extend the length of each path upwards by a factor of 2 (i.e. extend a path of length $l$ up by $l$ levels), unless, of course, we hit the root while going up. The extension is called the *ladder*. See Figure 2.

Instead of storing an array with a path, we store an array with the path plus the ladder. The space is still linear, because the ladder can be amortized against the path. However, queries can now be answered in $O(\lg n)$ time. Indeed, assume the current path is of length $l$, and the node at the top of the ladder is $w$. Then, HEIGHT($w$) $\geq 2 \cdot l$. Otherwise, the longest path containing $w$ would have continued down with the laddert and our the current path. Therefore, each step either doubles the size of the path, or finishes.

**Combine ladder decomposition & jump pointers.** The idea is that jump pointers start with large jumps (that become exponentially smaller), and the ladder decomposition starts with small jumps (that become exponentially larger). Then, we can combine these and have one big jump with jump pointers and another big jump with ladders. A query proceeds as follows:

- take one jump pointer to go up $\lfloor \lfloor l \rfloor \rfloor > l/2$ nodes. Call the intermediate node that is reached $w$. We have $\text{HEIGHT}(w) > l/2$, because we know there is a path of length $\lfloor \lfloor l \rfloor \rfloor$ below $w$.
- take one ladder step. Because $\text{HEIGHT}(v') > l/2$, we know that the ladder of $v'$ extends at least $l/2$ above $v'$ (or it includes the root), so we can get to the correct ancestor right away.

This strategy gives constant-time queries, but $O(n \lg n)$ space for the jump pointers.

**Tune jump pointers.** We want to store jump pointers only on leaves to reduce space. To accommodate that, we can store an arbitrary *leaf descendant* of every non-leaf node. The depth-$d$ ancestor of $V$ is the same as the depth-$d$ ancestor of its leaf descendant, so we can start queries at leaves. The space usage is $O(n + L \lg n)$ where $L$ is the number of leaves.

**Microtree/macrotree decomposition.** We want to limit the number of leaves, so we do a microtree/macrotree decomposition. The macrotree has $O(n/\lg n)$ leaves, so is takes $O(n)$ space using the solution from above.

Microtrees have $O(\lg n)$ branching nodes, so we can use a more brute-force approach. We first number nodes by the Euler tour of the microtree. Note that for each possible depth within the microtree, there are at most $O(\lg n)$ nodes at that depth (because there are at most $O(\lg n)$ paths from the restriction on branching nodes). So we can store a fusion structure for each possible depth, holding the Euler tour indices of the nodes on that depth. The total space is linear in the number of nodes. To find the ancestor of depth $d$ of node $V$, we just query for the predecessor of $V$ in $d$'s atomic heap.

**Weighted level ancestor.** A more general version of the problem involves edges weights. This is useful because for compacted tries, an edge may represent more than one letter. The solution for microtrees from above doesn't quite work, but similar ideas can be applied obtain $O(\lg \lg n)$ query time and $O(n)$ space. The $O(\lg \lg n)$ bound comes from predecessor search.

## References


[1] Stephen Alstrup, Jacob Holm. *Improved Algorithms for Finding Level Ancestors in Dynamic Trees.* ICALP 2000: 73-84

[2] Michael A. Bender, Martin Farach-Colton. *The Level Ancestor Problem simplified.* Theor. Comput. Sci. 321(1): 5-12 (2004)

[3] Omer Berkman, Uzi Vishkin. *Finding Level-Ancestors in Trees.* J. Comput. Syst. Sci. 48(2): 214-230 (1994)

[4] Richard Cole, Lee-Ad Gottlieb, Moshe Lewenstein. *Dictionary matching and indexing with errors and don't cares.* STOC 2004: 91-100

[5] Paul F. Dietz. *Finding Level-Ancestors in Dynamic Trees.* WADS 1991: 32-40