CSE 584A Class 6

Jeremy Buhler

February 5, 2018

1 A New Paradigm for Exact Matching

All the algorithms we've seen so far must scan the entire text to find all occurrences of a pattern.

- Is linear-time performance in text size always acceptable?
- What if we expect to do *many* queries on same text? Do we really want to reread it each time? (e.g. performing many searches on GenBank, or against some genome)
- As a special case of above problem, what if we seek similarities between a text and *substrings* of a pattern? What if the "pattern" is actually another genome / database?
- *Example*: given two genomes, find *all* exact substring matches between them of length at least 25.
- (Might be used to help decide which regions of these genomes are good candidates for further alignment, or to estimate their similarity directly *without* alignment.)
- *Example*: given two sets of reads from two metagenome sequencing experiments, find *all* exact substring matches between them of length at least 16.
- (Might be diagnostic for occurrences of common organisms, such as certain bacteria or viruses, between two geographically diverse samples)
- In other cases, we seek repeated sequences in the text itself.
    - duplicated genes (e.g. TCR and immunoglobulin genes)
    - interspersed repetitive elements
    - overlapping reads for assembly
- In simplest case, we seek pattern matches of a fixed length (k-mers).
- More generally, the sizes of the matches we seek are *not* fixed *a priori*.

Formally, let's consider two closely related problems.

- Can we solve the exact matching problem on a text T in worst-case time $o(|T|)$?
- **Substring Matching**: given two texts T_1 and T_2, find all pairs of identical substrings $t_1 \in T_1$ and $t_2 \in T_2$ of length at least k.

What can we say about these problems?

- If $P = a^n$ and $T = a^m$, output size for exact matching problem is $\Theta(m) = \Theta(|T|)$. Hence, we cannot do better.
- If $T_1 = a^{m_1}$ and $T_2 = a^{m_2}$, and $k = O(1)$, output of identical k-substring problem has size $\Theta(m_1 m_2)$.
- In the worst case, simply printing the problem's answer has a large cost, no matter how it is solved.
- Rather than admit defeat, let's consider an alternative way to estimate costs.
- Divide cost of exact matching into two parts $W(|P|, |T|) + E(P, T)$. The *search time* $W(|P|, |T|)$ is cost of searching for matches, while the *output time* $E(P, T)$ is (data-dependent) cost of enumerating any matches found.
- In general, enumeration time should be $O(\#$ matches). If we explicitly list all matches as our output, this O is really a Θ.
- While cost $E(P, T)$ is data-dependent, cost $W(|P|, |T|)$ depends on particular search algorithm used.
- For example, KMP algorithm has cost $W(|P|, |T|) = \Theta(|P| + |T|)$, whether or not it finds *any* matches!
- In practice, we assume that $E(P, T)$ is small (else you asked the wrong question) and attempt to minimize the search time $W(|P|, |T|)$.

2 Preprocessing is OK

Consider the exact matching problem. We knew we wanted to compare the pattern against lots of text, so it was OK to spend time learning about pattern's structure to accelerate the search.

- If we're going to search against the same text multiple times, should we be allowed the same freedom to preprocess the text?
- If you put up a web server that processes 10^5 searches a day, wouldn't you spend a few hours processing your genome database first to make sure those searches run as fast as possible?
- (NIH already fielded that many BLAST searches per day on GenBank back in 2004.)
- Moreover, if you want to do short read mapping, you're going to have to map tens to hundreds of millions of reads back to the same reference sequence!
- Hence, let's allow ourselves the luxury of *indexing* the text (preferably in time $O(|T|)$, definitely in time $o(|T|^2)$) to ensure that subsequent searches run in time $o(|T|)$.

3 Sublinear-Time k-mer Matching with Hashing

What are some "obvious" ways to achieve sublinear-time exact matching?

- Let's consider the hoary old k-mer matching problem.
- Given two sequences T_1, T_2, find all substring matches between T_1, T_2 of length exactly k.
- *Idea*: build a hash map of all k-mers that occur in T_1, mapping each k-mer to a list of positions where it occurs.
- Now, for each k-mer in T_2, check whether it appears in the hash table. If so, enumerate its occurrences.
- With a decent hash table, "typical" performance of this method is constant time per hash insert and lookup, so $W(|P|, |T|) = \Theta(|P| + |T|)$.
- (But worst case is much worse!)
- We can try to get *worst-case* linear time by designing a collision-free hash function given T_1 and using that to build the table. See results on *minimal perfect hashing.*
- *Example* of MPH: Sailfish tool for RNA isoform quantification (Patro, Mount, and Kingsford)
  `http://www.nature.com/nbt/journal/v32/n5/full/nbt.2862.html`

Why not stop here?

- We may not be satisfied with fixed k.
- Also, hash tables are generally quite space-inefficient. Each base of sequence hashed translates into 10-20 bytes of memory.
- *Example*: AAF alignment-free, assembly-free phylogeny reconstruction tool (Fan et al.)
  `http://bmcgenomics.biomedcentral.com/articles/10.1186/s12864-015-1647-5`
- Hashing all the k-mers needed to compare short read sets from two primate-sized genomes took 40-70 gigabytes of RAM!
- *Important lesson*: when indexing very large string data sets, space, and in particular the constant factor (bits or bytes per character) on space, matters.

4 Sublinear-Time Exact Matching with Suffix Array

- Another indexing strategy: the *sorted suffix array*
- Let $T[i..|T|]$ denote the ith suffix of T.
- Construct an array A of $|T|$ pointers, such that the ith pointer points to suffix $T[i..|T|]$.

- Now *sort* the array A lexicographically by the pointed-to strings.
- **Example**:

- Clearly, can represent array with just one pointer per suffix, so $\Theta(|T|)$ space overall (constant is 4-8 bytes/base).
- Naively, constructing A from T requires $\Theta(|T| \log |T|)$ *string* comparisons if we use standard comparison sort.
- Number of *chars* compared could be $\Theta(|T|^2)$ in worst case. Think a^m. More realistically, think of a genome with a large number of repeated substrings.
- To find all occurrences of pattern P in T, must find all suffixes of T that begin with P.
- (These suffixes form a contiguous range in A.)
- Requires two binary searches with $O(\log |T|)$ string comparisons apiece (one to find first match, one to find last).
- Worst-case cost is $\Theta(|P| \log |T|)$ character comparisons.

Building and search cost isn't bad for *random* seqs, but it is very sensitive to the level of repetition in real sequences. DNA is particularly bad because the alphabet is so small. We'll discuss enhancements that let us do better at both building and search for suffix arrays.

- *Aside*: we assume above that we are using a comparison sort.
- Given a fixed alphabet Σ, why not use radix sort?
- (Counting sort pass on each character position or q adjacent positions.)
- If we are sorting n k-mers, this takes time $\Theta(k(n + |\Sigma|))$, which is pleasantly linear in n.
- But to sort the suffixes of T, we wind up spending time proportional to $\sum_i |T[i..n]| = \Theta(|T|^2)$.
- (This is true even if we avoid having to pad short suffixes out to the length of the longest suffix.)

5 Sublinear-Time Exact Matching with Suffix Tree

Are there alternative indexing strategies that let us do even faster search?

- Suppose we took our sorted suffix array and merged the common parts of adjacent strings?
- **Example**:

- Resulting structure is a *tree*, not an array.
- This data structure is called a *suffix tree*.

A little more formally…

- Let S be a string of length m over alphabet Σ.
- The suffix tree τ_S for S is a rooted tree whose edges are *labeled* with substrings of S.
- For any node x of τ_S, let the *node label* of x, denoted *label*(x), be the concatenation of the strings labeling the edges from the root down to x.
- τ_S has m leaves, each with an unique index between 1 and m.
- For leaf ℓ_i, the node label *label*(ℓ_i) is exactly the ith suffix S_i of S. Leaves are arranged in lexicographic order of their suffixes.
- An internal node x with label α occurs in T iff two suffixes of S both start with the same substring α but differ in their next character.
- Hence, each internal node has degree at least 2 and at most $|\Sigma|$.

How big is the suffix tree for a string?

- Tree starts with a single root and ends with exactly m leaves.

- Hence, there must be at most $m-1$ internal splits, each of which is represented by an internal node.

- Conclude that suffix tree for S has at most $2m$ total nodes.

- What about labeling edges?

- Each edge is labeled with a contiguous substring of S.

- May represent this label as (i, k), where i is the start of the label in S, and k is its length.

- Takes only constant space per edge (hence per node).

- Conclude that suffix tree also uses $\Theta(|T|)$ space.

- Constant factor is significant: a pointer for every edge, and $\Theta(|\Sigma|)$ pointers in every node. If not careful, space may exceed 50 bytes/character even for small alphabets!

- *Questions for later*: how fast can you build a suffix tree from a string? From its suffix array?

Given tree τ_S, how fast can we do exact matching of pattern P in S?

- *Algorithm*: start at root of tree. Trace path corresponding to P.

- If we cannot trace the full length of P, then P is not a prefix of any suffix of S, and hence does not occur in S.

- Otherwise, suppose the path for P ends on some edge e, and let node x be the lower endpoint of e.

- For every leaf ℓ_i in the subtree below x, P is a prefix of suffix S_i and hence matches $S[i..i+|P|-1]$.

- Hence, we can locate all matches to P in S in time $\Theta(|P|)$.

- Moreover, given node x, we can enumerate all matches in time proportional to their number by an in-order tree walk starting at x.