CSE 584A Class 19

Jeremy Buhler

April 4, 2018

1 Benefits of Suffix Trees for Speeding Up Matching

- Recall how we match a pattern $P[1..m]$ against a suffix tree τ for text T.
- We do a top-down traversal of τ to find a subtree whose leaves represent all suffixes of T starting with P.
- If traversal is at node in τ, take edge corresponding to next character of P.
- Otherwise, match characters of P along the current edge until we reach next node.
- If we cannot proceed, matching fails.
- Picture:

- Why is this approach efficient?
- Suffix tree merges common prefixes of every suffix.
- Hence, traversing an edge effectively performs matching computation vs every suffix below this edge at once!

We want to take advantage of the "work compression" provided by a suffix tree for inexact matching as well.

- Which inexact matching problem are we solving?
- Two principal problems of interest.
- **Problem 1**: perform *semi-local* alignment

- Must match *all* of pattern P to any substring(s) of T.
- *Typical use*: determine where a short read occurs in T, with a few differences allowed.
- **Problem 2**: perform *fully local* alignment
- Must match a *substring* of P to any substring(s) of T.
- *Typical use*: any BLAST-like problem, i.e. given a fairly long query q, where does q locally align well to T?
- Which and how many alignments should we return?
- Again, two possibilities.
- **Result 1**: return every alignment of P to T that scores above some threshold $\theta \geq 0$.
- **Result 2**: return the best k distinct alignments (or fewer, if fewer than k have a positive score).
- May combine these criteria, e.g. best k alignments, or fewer if fewer than k have score $\geq \theta$.
- Note that each distinct alignment of P may occur at multiple locations in T, but each location has an identical alignment path and aligns the same substring of P to identical strings in T.

2 Extending Dynamic Programming Alignment to a Tree

All of the problem/result combinations described above can be formulated as variants of dynamic programming alignment on the tree τ for T.

- Consider aligning P to two strings T_1 and T_2 that share a common prefix of length ℓ.
- We could use (linear or affine) Smith-Waterman for fully local alignment, or modified Needleman-Wunsch for semi-local alignment.
- In either case, we fill in matrices of size proportional to $|P| \times |T_1|$ and $|P| \times |T_2|$.
- **Picture**:

- Now the first $\ell + 1$ rows of the matrices for each problem are identical, since we do the same computation for each!
- Hence, if we merge the common prefix of T_1 and T_2, we could share the portion of the matrix once up to the point where they diverge, then duplicate the last row and continue computing separately for each suffix.

- **Picture**:

- Let's extend this idea to aligning vs *all* suffixes of a string T.
- Form suffix tree τ from T.
- Apply DP alignment top-down to every branch of τ, starting from root and copying the last row of the DP matrix to all child edges at each branch point.

3 Virtual Tree Traversal with the BWT

The tree traversal version of DP is straightforward given a real suffix tree τ, but such a tree is quite space-intensive!

- **Idea**: generalize the BWT-based exact matching algorithm to perform "virtual" traversal of the tree.
- First, observe that every position in tree τ (node or not) can be described by a depth plus a set of leaves below that position.
- Moreover, consecutive leaves in τ correspond to consecutive suffixes in the suffix array A for T.
- Hence, we can represent our position in τ by a depth plus the endpoints of a consecutive range of suffixes in A.
- Root of τ (depth 0) corresponds to the set of *all* suffixes $[1..|T|]$.
- Moving down one character in τ either keeps the same range of suffixes (if we are in the middle of an edge of τ) or narrows the range (if we follow one child edge out of a node of τ).
- When the next downward step causes the list of suffixes below us to become empty, we've reached a leaf.
- When our downward traversal stops, the final position immediately gives us the list of all suffixes whose beginnings match up to the final depth d.

We're going to show how to use the BWT to update our position in τ by one downward step in constant time for steps within an edge, or $O(|\Sigma|)$ time for steps across an internal node.

- Recall that, given the BWT of text T, if we know the range of suffixes in the SA matching a string S, then in constant time we can find the range matching string cS, for $c \in \Sigma$.
- Now suppose we build the BWT and SA from the *reverse* of T, i.e. T^R.

- With this construction, if we know the range of suffixes in the SA for T^R matching a string S^R, then in constant time we can find the range matching string $cS^R = (Sc)^R$.
- Conceptually, we are descending the suffix tree τ^R for T^R using a reversed pattern P^R.
- Now suppose the traversal of τ^R stops at some depth d with interval $[i \dots j]$ in the SA $\tilde{A}$ for T^R.
- As we observed earlier, if $\tilde{A}[i] = k$, then $T^R[k..n]$ starts with some string S^R, with $|S^R| = d$.
- Hence, $T[1..n-k+1]$ ends with string S (assumes 1-based indices!).
- Conclude that, for every suffix in the final interval $\tilde{A}[i..j]$ starting with S^R, there is a corresponding occurrence of S in T, and we can locate it in constant time.

Great, but how is this related to DP alignment?

- We will align P^R to T^R, which gives same alignments (reversed) as aligning P to T, whether they are fully local or semi-local using all of P.
- Suppose the DP has progressed down some path in τ^R labeled with string U^R.
- To fill in next row, we first compute tree positions corresponding to strings $cU^R = (Uc)^R$ for every $c \in \Sigma$, using the BWT.
- If we are in the middle of an edge, only one position (say, the one for $(Ua)^R$) will have a nonempty s.a. interval, so we continue the current DP with a as the next character of the text.
- if we are at a node, two or more positions will have nonempty intervals. In this case, we duplicate the last row of the DP matrix once per possible continuation and separately continue the DP for each one.
- At the end of the algorithm, we need to translate the coordinates of any alignments found from P^R/T^R to P/T, which is trivial given the lengths of P and T.

4 What Are We Returning?

- If the goal is to find all alignments of P in T of score $\geq \sigma$...
- We must complete the DP computation on the whole tree.
- To find matches involving *all* of P, we descend each possible tree path until we've aligned all of P (or hit a leaf) and remember the paths with high enough scores.
- To find matches involving *substrings* of P, we again descend each possible path (this time, allowing restarts in P) but remember the locations of cells scoring above σ.
- (*Note*: don't actually do the latter – we'll discuss a better way next time.)

But what if the goal is to find the single *best* alignment of P to T?

- Recall our formulation of Needleman-Wunsch as a weighted alignment *graph*
- In this problem, we start at the starting vertex, and the aim is to find a path from the start to a certain goal vertex with the highest total score.
- In AI, this is a *state-space search* problem.
- To find the goal, a basic strategy is to maintain a frontier of all the vertices from which we can expand, sorted by the score of the best path to them from the start.
- If we always explore edges out of the best-scoring vertex on the frontier first, we are guaranteed to find a best-scoring path to the goal before finding any other such path.
- (And if we keep going after finding the goal, we enumerate paths to it in non-increasing order of their score.)

Consider extension to tree alignment...

- In this case, there are multiple goal vertices – any alignment path that uses all of P is fine.
- Moreover, the graph is now branching according to the structure of the tree:

- But the same basic idea works!
- Note that the next vertex to expand at a given moment may be on any path down the tree – we might jump back and forth in successive steps.
- The default mode of Bowtie and BWA is to find all the best-scoring matches for each read, so they use this strategy, rather than exploring the entire tree.

Next time: a variety of sneaky tricks for speeding up the search process by pruning low-scoring paths!