CSE 584A Class 5

Jeremy Buhler

February 3, 2016

1 Dictionary Matching

- Let $D = \{P_1, P_2, \ldots P_k\}$ be a *dictionary* of k different pattern strings.
- **Problem**: how fast can we find all matches to *any* pattern in D in a text T?
- Naively, if we search for every pattern individually, we make k passes over the text, so cost is $\Theta(\|D\| + k|T|)$, where $\|D\| = \sum_{j=1}^{k} |P_j|$.
- Even if we run all k KMP automata concurrently, so that we need read the text only once, we still do $\Theta(k)$ work per character read (for real if we use the g links, or in an amortized sense if we use the f links).
- Let's see if we can improve on this time!

Idea: merge the various KMP automata into a *single* automaton that can track partial matches to *every* pattern at once.

- **Important assumption**: no pattern in D is a substring of any other pattern (*you will fix this in your homework*).
- First hack we need is a way to merge all the strings into one structure.
- For a string P, let its *state sequence* be the expansion of P into a path of $|P| + 1$ states, such that transition from ith to $i + 1$st state is labeled with $P[i]$.
- **Example**:

- **Defn**: a *trie* τ for a dictionary D is a tree structure obtained by constructing the state sequences for each P_j in D and then merging their common prefixes.
- **Example**:

- A trie consists of states and edges (transitions). For state q of a trie, the *label* of q is the string formed by contatenating the characters on the path from the root down to q.
- **Defn**: a trie τ matches text T starting at position ℓ if a string $T[\ell \ldots]$ is the label of some leaf state.
- There are k leaf states corresponding to the k patterns in D. We mark all leaf states as "accepting" and annotate each with its corresponding pattern.
- We can easily turn D into a trie in total time $\|D\|$:
  - Start with trivial trie consisting of a single root state q_0.
  - For each pattern P_j, walk down the path starting at q_0 that is labeled with the characters of P_j, until we cannot continue at some state q.
  - Create a new branch off of q and append the state sequence for the rest of P_j.

2 Efficient Trie Matching

Naive trie matching in a text is very similar to naive string matching: attempt to match the trie starting at each text position. Worst-case cost is $\Theta(\max_D |T|)$ comparisons, where $\max_D$ is the length of the longest pattern in D. Can we do better?

- We can leverage principles similar to KMP to avoid redundant comparisons!
- **Defn**: for state q of a trie τ, let sp_q be the length of the longest proper suffix of the string labeling the path from the root to q that also labels a path starting at q_0.
- In other words, it's the longest suffix-prefix match for the trie path ending at q.

- **Claim**: suppose we attempt to match a trie τ starting at $T[1]$, and the attempt terminates at state q (so that we cannot continue down any path without a mismatch).
- Let $d(q)$ be the depth of q in τ (the root is depth 0). Then no match to τ occurs starting in $T[2..d(q) - sp_q]$.
- **Pf**: exactly analogous to proof of correctness for KMP.

- Note also that, as in KMP, we can skip the first sp_q characters after moving the trie forward by $d(q) - sp_q$ in T, because we know that they label a prefix of some path. Matching can continue from the end of that path.
- But *which path is it*?
- **Defn**: the *failure link* f_q is an edge pointing from state q to the state at depth sp_q whose label matches the corresponding suffix of q's label.
- If we cannot continue from state q, we resume matching from state f_q.
- (If we cannot continue from the root, we stay there, so $f_{q_0} = q_0$.)
- As for KMP, matching in T resumes at the character that caused the mismatch, or after that character if the mismatch was at q_0. As before, on success, we resume after the last match.

This gives us the following search algorithm:

```
MatchTrie(T, τ)
  q ← q_0
  ℓ ← 1
  while ℓ ≤ |T| do
    while T[ℓ] labels a trie edge q → q' in τ do
      q ← q'
      ℓ++                                  ▷ be sure to stop if ℓ reaches |T|!
    if q is an accepting state for pattern P_j
      emit "match to P_j ending at ℓ"
    if q = q_0
      ℓ++                                  ▷ move past initial mismatch
    q ← f_q
```

- **Claim**: MatchTrie performs at most $2|T|$ reads of characters from T.
- **Pf**: exactly analogous to original KMP proof.
- Every mismatch takes a failure link, which moves the the trie forward in T. Hence, at most $|T|$ reads due to mismatches.
- Moreover, we never reread a match in T, so at most $|T|$ matches. QED

Conclude that, once we have the trie for D, we need only $2|T|$ comparisons to find all matches to D in the text, independent of k or $\|D\|$!

3 Efficient Construction of Failure Links

- Procedure for constructing failure links is exactly analogous to that for KMP.
- We can use failure links for states closer to the root to derive the links for states farther from the root.

computeF(τ)

$f_{q_0} \leftarrow q_0$
for each non-root state q of τ in breadth-first order **do**
    $r \leftarrow$ parent of q
    $c \leftarrow$ character labeling edge $r \rightarrow q$
    $r \leftarrow f_r$
    while $r \neq q_0$ and no trie edge out of r is labeled with c **do**
        $r \leftarrow f_r$

    if trie edge $r \rightarrow r'$ is labeled with c and $r' \neq q$
        $f_q \leftarrow r'$
    else
        $f_q \leftarrow q_0$

Note the "$r' \neq q$" in the last test; this is needed to handle the case that $r = q_0$, so that the edge $r \rightarrow q$ is labeled with c.

- **Claim**: given trie τ for dictionary D, can construct all failure links in worst-case time $\Theta(\|D\|)$.
- **Pf**: again analogous to building failure links for KMP.
- Each state q of τ corresponds to a prefix of some pattern $P_j \in D$ (perhaps more than one pattern if we've merged common prefixes).
- For state q, the best suffix-prefix match interval is $P_j[d(q) - sp_q + 1 \ldots d(q)]$.
- When we compute f_q, we try to extend the interval in P_j for q's parent. Each time we fail to so extend, we follow a failure link and move the start of the interval for P_j forward by at least 1 character.
- Hence, total number of failure links followed is at most $\|D\|$, the total length of all patterns in D.

Conclude that, given dictionary D and text T, we can find all matches to D in T in total time $\Theta(\|D\| + |T|)$, independent of number of patterns in D.

4 Extensions

- We can define enhanced failure links $g_{q,c}$ just as we did for KMP to ensure that we read each character of T only once, at the cost of increasing space and construction time by a factor of $|\Sigma|$.
- What if some patterns in D are substrings of other patterns?
- *Easy case*: patterns can occur as prefixes of another pattern, but not otherwise.
- (*Hint*: mark the states corresponding to the ends of such patterns.)
- *Hard case*: patterns can occur as arbitrary substrings of another pattern.
- (*Hint*: you may have to emit matches to multiple strings when reaching an accepting state! Challenge is to find and mark all accepting states while maintaining same asymptotic construction time *and space* for automaton.)

- Note that, if one pattern can be a substring of another, the total number of matches between D and T is no longer $\leq |T|$.
- **Ex**: $T = a^n$, $D = \{a, aa, \ldots a^m\}$ yields $\Theta(mn)$ matches.
- Hence, we must add a term to the search cost to account for the total number of matches emitted by the search.

With the extension to allow arbitrary substring relationships among patterns, this efficient trie matching procedure is the well-known *Aho-Corasick algorithm*. It has applications to large-scale genomics even today. For an example involving a dictionary with thousands of patterns, see Pizzi, Rastas, and Ukkonen, "Finding significant matches of position weight matrices in linear time", *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 8:69-79, 2009.