CSE 584A Class 3

Jeremy Buhler

January 27, 2016

1 Improving On Z-box-based String Matching

The Z-box algorithm assumes that we store *all* the z-values prior to position i before computing z_i. Without this assumption, the algorithm doesn't necessarily work (consider computing z-values for a^m).

- For pattern-matching, the Z-box algorithm uses $P \cdot T$. We really only care about the structure inside P, yet we apparently need space proportional to $|P| + |T|$.
- Can we reduce the working space to $\Theta(|P|)$, independent of $|T|$?

Moreover, we haven't yet (directly) exploited the intuition we developed at the beginning of this whole process: *with the right preprocessing of P, we can avoid some of the character comparisons performed by the naive algorithm.*
Example:

- Suppose we check P versus T at offset 1.
- Match fails at 7th char.
- Should we advance P by 1 position, as in naive algorithm?
- Clearly NOT: match would fail immediately! (a versus c)
- After checking P at offset 1, we *know* that $T[2] = c$ (because it matched $P[2]$), so there's no excuse for not skipping it.
- In fact, we could skip all the way to $T[5]$ and resume comparing there.

There are two common algorithm families that use this kind of information.

- Fastest in practice is Boyer-Moore and related methods, e.g. Apostolico-Giancarlo.
- Other common algo is Knuth-Morris-Pratt, which I find easier to understand from scratch.

2 Intro to KMP

First, let's formalize the observation that we can skip some sequence positions.

- **Defn**: Let S be a string. Let $sp_i(S)$ be the length of the longest *proper suffix* of $S[1..i]$ that matches a prefix of S.
- (By this definition, $sp_1(S) = 0$.)
- **Example**:

- Suppose P matches $T[1..i]$ but mismatches at $T[i+1]$.
- By definition, $P[i - sp_i + 1..i]$ is the same as $P[1..sp_i]$.
- But $P[i - sp_i + 1..i]$ is also the same as $T[i - sp_i + 1..i]$.
- So, let's *shift* P forwards until $P[1..sp_i]$ aligns with $T[i - sp_i + 1..i]$ and keep going.
- Do the math... P shifts forward by exactly $(i - sp_i + 1) - 1 = i - sp_i$ positions.
- P *might* match T at the new offset. We know that its first sp_i characters *definitely* match.

Is this shift safe? How do we know we didn't skip over another copy of P in T?

- **Claim**: if $P[1..i]$ matches $T[1..i]$, then shifting P by $i - sp_i$ positions never skips an occurrence of P in T.
- **Proof**: Suppose we *did* miss an occurrence of P in T.
- Such an occurrence starts at some offset $T[j]$ between $T[2]$ and $T[i - sp_i]$, inclusive.

- We know that $T[j..i] = P[j..i]$, since all these positions are inside the matching region.
- But if a match starts at $T[j]$, then $T[j..i] = P[1..i-j+1]$.
- Conclude transitively that $P[j..i] = P[1..i-j+1]$.
- Hence, $sp_i(P) \geq i - j + 1$, by definition of $sp_i(P)$. (That is, we've proven a *long* overlap of P with itself.)
- But we assumed at the start that $j \leq i - sp_i(P)$, so $sp_i(P) \leq i - j$! (That is, the new occurrence had to start *before* the position where we would have shifted P.)
- **Contradiction**. QED.
- (Similar argument holds at *any* offset into T.)

3 KMP Algorithm

We now have a nifty search algorithm.

```
KMP(P, T)
    compute sp_i(P) for 1 ≤ i ≤ |P|
    k ← 1                                   ▷ current start of pattern in T
    q ← 0                                   ▷ current offset into pattern
    while k ≤ |T| − |P| + 1 do
        explicitly compare P[1 + q...] to T[k + q...]
        if end of P reached w/no mismatch
            emit "match at k"
        if P[1] ≠ T[k]                      ▷ initial mismatch
            k ← k + 1
            q ← 0
        else
            j ← position in P of last matching char
            k ← k + j − sp_j(P)
            q ← sp_j(P)
```

Example:

How fast is this algorithm?

- **Claim**: KMP's comparison loop performs at most $2|T|$ total character comparisons.

- **Proof**: We shift P forward whenever we encounter a mismatch or the end of P.
- When shifting on a match to P, we don't reread any text chars (if previously at $T[k]$, code restarts comparisons at $T[k + |P|]$).
- When shifting on a mismatch, we may reread the mismatched text char, but no other (we restart comparisons from $T[k+j]$, the site of the mismatch, or immediately *after* it if $P[1]$ mismatches).
- Conclude that we reread at most one text char per shift.
- We therefore read at most $|T| + s$ text chars, where s is number of shifts.
- But every shift moves k forward by at least one character in the text, so $s \leq |T|$.
- Conclude that we perform at most $2|T|$ character comparisons to T.

4 Computing the sp_i's

So, how do we compute the values sp_i? It wouldn't matter much if it were expensive, since P is small, but in fact it can be done using only the Z-values, in time $\Theta(|P|)$.

- For each position i of P, we want the longest prefix of P that matches a proper suffix of $P[1..i]$.
- Suppose a Z-box starts at some position $j \leq i$ and overlaps i. Then $P[j..i]$ matches a prefix of P by definition!

- So, for each i, we need to find the Z-box (if any) overlapping i that starts *furthest to its left*.
- If that Z-box starts at position j, then $sp_i = i - j + 1$.
- If no Z-box covers j, then $sp_i = 0$.

How can we do this quickly?

- Every time we find a Z-box, set sp_i values for all positions inside it. (Decreases from right to left.)
- This *only* works if we process Z-boxes in right-to-left order by their starting points. If we go left to right, we'll find shorter suffixes *after* longer ones.
- Also, multiple Z-boxes can cover one position, so no guarantee of constant work per position (and hence linear time overall).
- **Solution**: first pass marks *only* indices where some Z-box ends. (Remember which Z-box!)

- In second pass, go right to left again. Each time we see a mark, determine (in $O(1)$ time) if the Z-box Z_j we're about to enter starts further left than the one we're in (if any). If so, switch to computing sp values from Z_j.
- Conclude that computation of sp values takes time $\Theta(|P|)$.

Example: