CMSC423: Bioinformatic Algorithms, Databases and Tools
Lecture 7

Exact string matching
Suffix trees
Suffix arrays
Basic idea: 1-D dynamic programming

Can \( Z[i] \) be computed with the help of \( Z[j] \) for \( j < i \)?

Assume there exists \( j < i \), s.t. \( j + Z[j] - 1 > i \) then \( Z[i - j + 1] \) provides information about \( Z[i] \)

If there is no such \( j \), simply compare characters \( T[i..] \) to \( T[0..] \) since they have not been seen before.
Three cases

Let \( j < i \) be the coordinate that maximizes \( j + Z[j] - 1 \)
(intuitively, the \( Z[j] \) that extends the furthest)

I. \( Z[i - j + 1] < Z[j] - i + j - 1 \) => \( Z[i] = Z[i - j + 1] \)

II. \( Z[i - j + 1] > Z[j] - i + j - 1 \) => \( Z[i] = Z[j] - i + j - 1 \)

III. \( Z[i - j + 1] = Z[j] - i + j - 1 \) => \( Z[i] = ?? \), compare from \( i + Z[i - j + 1] \)
Time complexity analysis

• Why do these tricks save us time?

1. Cases I and II take constant time per Z-value computed – total time spent in these cases is $O(n)$
2. Case III might involve 1 or more comparisons per Z-value however:
   - every successful comparison (match) shifts the rightmost character that has been visited
   - every unsuccessful comparison terminates the “round” and algorithm moves on to the next Z-value

   total time spent in III cannot be more than # of characters in the text

Overall running time is $O(n)$
Space complexity?

- If using Z algorithm for matching, how many Z values do we need to store?

```
PPPPPPPPPPP$TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
```

- Only need to remember Z-values for pattern and the “farthest reaching Z-value” (Z[j] in what we discussed before)
Z algorithm, not just for matching

- Lempel-Ziv compression (e.g. gzip)

If \( Z[i] = 0 \), just send/store the character \( T[i] \), otherwise, instead of sending \( T[i..i+Z[i] – 1] \) (\( Z[i] – 1 \) characters/bytes) simply send \( Z[i] \) (one number)

- Note: other exact matching algorithms used for data compression (e.g. Burrows-Wheeler transform relates to suffix arrays)
Knuth-Morris-Pratt algorithm

Given a Pattern and a Text, preprocess the Pattern to compute $sp[i] =$ length of longest prefix of $P$ that matches a suffix of $P[0..i]$

Compare $P$ with $T$ until finding a mis-match (at coordinate $i + 1$ in $P$ and $j + 1$ in $T$). Shift $P$ such that first $sp[i]$ characters match $T[j – sp[i] + 1 .. j]$. Continue matching from $T[i+1]$, $P[sp[i]+1]$
Boyer-Moore algorithm

Preprocess the pattern, computing, for every $i$, $L[i] =$ largest coordinate $< n$, s.t. $P[i..n]$ matches a suffix of $P[1..L[i]]$ (inverted $Z$ function)

Match the pattern backwards (starting at the right) until mismatch. Shift the pattern such that $P[L[i] – n + i + 1]$ matches at $T[j]$ Repeat.

Bad character rule: find character $T[j – 1]$ in $P$ and shift until it matches. Choose the longest shift (btwn. suffix & char. rules)
Suffix trees
Intro to suffix trees

• Used in fast exact matching
• Basic idea: extend a trie – structure for storing multiple strings
Suffix tree

- Extends trie of all suffixes of a string

1. ATCATG
2. TCATG
3. CATG
4. ATG
5. TG
6. G
Suffix tree ...cont

- To store in linear time – just store range in sequence instead of string
- To ensure suffixes end at leaves, add $ char at end of string
- ATCATG$
Suffix links

- Link every node labeled aS for some string S to node labeled S (note – it always exists)
Suffix trees for matching

- Suffix trees use $O(n)$ space
- Suffix trees can be constructed in $O(n)$ time
- Is CAT part of ATCATG?
- Match from root, char by char
- If run out of query – found match
- Otherwise, there is no match

intuition: CAT is the prefix of some suffix
Suffix links – useful for substring matches

- Does any part of AGATG match string AGCAGT?