String matching and searching

- String: sequence of characters from a fixed alphabet
  - Text, dictionaries: 26 letters + special symbols
  - DNA strands: 4 letters
  - Proteins: 20 letters (aminoacids)

- Typical problems:
  - Find pattern P in text T
  - Find longest common substring of strings T₁ and T₂
  - Given string T, find substrings P that occur more than k times
  - …

String algorithms

- Knuth-Morris-Pratt algorithm
  - find pattern P in string T
  - preprocess P (construct Fail array)
  - search time: O(n+m) \( n=|T|, m=|P| \)

- Data structures for strings
  - trie
  - suffix tree & generalized suffix tree
  - suffix array

- Many applications
  - e.g. computing common longest substring

Trie data structure

- Data structure for storing a set of strings
- Name comes from “retrieval”, usually pronounced as “try”

- Rooted tree
- Edges labeled with letters
- Leaves correspond to input strings

\{ had, held, help, hi \}
Trie data structure

• Cannot represent strings T and TT
• Solution: append special symbol $ to the end of each word

{ had, held, help, hi }
Trie data structure

- Application: storing dictionaries
- Space: $O(m_1 + ... + m_k)$, $m_i$ = length of string $T_i$
- Looking up word of length $n$: $O(n)$
  - assuming alphabet has constant size

- Alternative to binary trees
  - pros and cons (see wikipedia)
  - all nodes store items, not just leaves
  - unlabeled edges

Tries for storing suffixes

- This lecture: use tries for storing the set of suffixes of string $T$

Reducing space requirements

- Trick 1: Delete non-branching nodes, merge labels (compact trie)
Reducing space requirements

- # leafs: n+1
- # internal nodes: at most n
- # edges: at most 2n edges
- Trick 2: Store edge labels via two pointers into the string (begin & end)

Suffix trees

- Called the *suffix tree* for string T \[\text{[Weiner'73]}\]
- D. Knuth: “algorithm of the year 1973”
- Can be constructed in linear time
  - e.g. [McCreight'76], [Ukkonen'95]
  - algorithms are complicated [will not be covered]
- Used for all sorts of string problems

Suffix trees for string matching

Does text T contain pattern P?

- Construct suffix tree for T in O(|T|) time
- Query takes O(|P|) time!
- same as Knuth-Morris-Pratt, but...
Suffix trees for string matching

- Scenario: text T is fixed, patterns P_1,...,P_k come one at a time
  - T is very long
- Time: O(|T| + |P_1| + |P_2| + ... + |P_k|)
- Knuth-Morris-Pratt can be generalized to multiple patterns (Aho-Corasick alg.). Same complexity as above, but requires knowledge of P_1,...,P_k in advance

Getting extra information

Task: Get all occurrences of P in T (give a list of start positions)

- Can be done in O(|P|+c) time where c is # of occurrences
  - Locate the node corresponding to P
  - Traverse all leaves of the subtree rooted at this node
Generalized suffix tree

- Input: a set of strings \( \{T_1, \ldots, T_k\} \)
- Generalized suffix tree: tree containing all suffixes of all strings
- Tree for \( \{abba, bac\} \):

```
\begin{align*}
abba$ & \\
ba$ & \\
ba & \\
\text{bba} & \\
\text{bac} & \\
\text{ac} & \\
\text{c} & \\
\text{\ldots} & \\
\end{align*}
```

Construction of generalized suffix tree for \( \{X, Y\} \)

- Construct suffix tree for \( X, Y \)
- Edges leading to red leaves are labeled as “\( \ldots, Y, \ldots \)”; delete “\( Y, \ldots \)”

```
\begin{align*}
abbaS_1 & \\
baS_1 & \\
ba & \\
\text{bba} & \\
\text{bac} & \\
\text{ac} & \\
\text{c} & \\
\text{\ldots} & \\
\end{align*}
```

Construction of generalized suffix tree for \( \{X, Y\} \)

- Construct suffix tree for \( X, Y \)
- Edges leading to red leaves are labeled as “\( \ldots, Y, \ldots \)”; delete “\( Y, \ldots \)”

```
\begin{align*}
abbaS_1 & \text{bacS}_2 \\
baS_1 & \text{bacS}_2 \\
ba & \text{bacS}_2 \\
\text{bba} & \text{bacS}_2 \\
\text{bac} & \text{bacS}_2 \\
\text{\ldots} & \\
\end{align*}
```
Construction of generalized suffix tree \( \{T_1, ..., T_k\} \)

- Can use the same trick
  - construct suffix tree for \( T_1 \ S_1 ... T_k \ S_k \)
- Linear runtime: \( O(|T_1| + ... + |T_k|) \)
- In practice, modify the algorithm (e.g. Ukkonen’s alg.) to get a slightly faster performance

Computing common longest subsequence of \( \{T_1, T_2\} \)

- Mark each internal node with red (blue) if it has at least one red (blue) child
- Nodes with both colors contain common subsequences

Computing common longest subsequence of \( \{T_1, ..., T_K\} \)

**Task:** For each \( r = 2, ..., K \) compute the length of the longest sequence contained in at least \( r \) strings

1. Construct generalized suffix tree for \( \{T_1, ..., T_k\} \)
Computing common longest subsequence of \( \{T_1, \ldots, T_K\} \)

**Task:** For each \( r = 2, \ldots, K \) compute the length of the longest sequence contained in at least \( r \) strings

1. Construct generalized suffix tree for \( \{T_1, \ldots, T_K\} \)
2. Mark each node \( v \) with color \( k = 1, \ldots, K \) if it has a leaf with that color
3. Compute \( c(v) = \# \) of colors at \( v \)
   - Naive computation: \( O(nK) \), \( n = |T_1| + \ldots + |T_K| \). Can also be done in \( O(n) \)

String corresponding to \( v \) is contained in exactly \( c(v) \) input strings

**Suffix array for text \( T[1..n] \)**

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{mississippi} & 0 & \text{e} & 11 & \text{i} & 10 \\
\text{ississippi} & 1 & \text{i} & 7 & \text{ippi} & 4 \\
\text{sissippi} & 2 & \text{issippi} & 4 & \text{ississippi} & 0 \\
\text{issippi} & 3 & \text{issippi} & 1 & \text{ississippi} & 0 \\
\text{ippi} & 4 & \text{ippi} & 1 & \text{issippi} & 2 \\
\text{ppi} & 5 & \text{ppi} & 2 & \text{issippi} & 3 \\
\text{pi} & 6 & \text{pi} & 3 & \text{issippi} & 4 \\
\text{i} & 7 & \text{i} & 4 & \text{issippi} & 5 \\
\text{e} & 8 & \text{e} & 5 & \text{issippi} & 6 \\
\hline
\end{array}
\]

**Suffix array:** stores suffixes of string \( T \) in a lexicographic order

**Pattern search from a suffix array**

**Task:** Find pattern \( P = \text{iss} \) in text \( T = \text{mississippi} \)

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{e} & 11 & \text{i} & 10 & \text{ippi} & 7 \\
\text{issippi} & 4 & \text{issippi} & 1 & \text{issippi} & 0 \\
\hline
\end{array}
\]

- All occurrences appear contiguously in the array
- Computing start & end indexes: binary search
- Worst-case: \( O(m \log n) \)
- In practice, often \( O(m + \log n) \)
- with smart implementation

\[\text{[Manber, Myers'90]: } O(m + \log n) \text{ worst-case}\]
- \( \text{store LCP table} \)
\[\text{[Abouelhoda et al.'04]: } O(m) \]
- \( \text{enhanced suffix array} \)
Enhanced suffix arrays

- [Abouelhoda, Kurtz, Ohlebusch, '04]
- Enhanced suffix arrays: array SA[0..n] + other tables
- In practice takes less space than suffix trees
- Every algorithm with a suffix tree can be converted to an algorithm with an enhanced suffix tree (with the same complexity)

Summary

- Suffix trees & suffix arrays: two powerful data structures for strings
- Space requirements:
  - suffix trees: linear but with a large constant, e.g. 20 |T|
  - suffix arrays: more efficient, e.g. 5 |T|
- Suffix arrays are more suited to large alphabets
- Practitioners like suffix arrays (simplicity, space efficiency)
- Theoreticians like suffix trees (explicit structure)

Further information on string algorithms

- Wikipedia (suffix trees, suffix arrays)