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 $T_1$ and $T_2$
  - Given string $T$, find substrings $P$ that occur more than $k$ times
  - ...

String algorithms

- **next**
  - Knuth-Morris-Pratt algorithm
- **time**
  - find pattern $P$ in string $T$
  - preprocess $P$
  - search time: $O(n+m)$, $n=|T|$, $m=|P|$

- **today**
  - 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
- Leafs correspond to input strings

```
{ had, held, help, hi }
```

Trie data structure

- Cannot represent strings $T$ and $T^*$
- Solution: append special symbol $\$ to the end of each word

```
{ had$, held$, help$, hi$ }
```

Trie data structure

- Cannot represent strings $T$ and $T^*$
- Solution: append special symbol $\$ to the end of each word

```
{ had$, held$, help$, hi$, h$ }
```

Trie data structure

- Cannot represent strings $T$ and $T^*$
- Solution: append special symbol $\$ to the end of each word

```
{ had$, held$, help$, hi$, h$ }
```
Trie data structure

• Application: storing dictionaries
• Space: $O(m_1 + \ldots + m_k)$, $m_i$=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)

Suffix trees

• Called the suffix tree for string $T$ [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
  \[ T \text{ is very long} \]
- 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 $\{T_1,...,T_k\}$
- Generalized suffix tree: tree containing all suffixes of all strings

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

- Construct suffix tree for $X \cup Y$
- Edges leading to red leaves are labeled as “...$S_1 Y S_2$”; delete “$Y S_2$”

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

- Construct suffix tree for $X \cup Y$
- Edges leading to red leaves are labeled as “...$S_1 Y S_2$”; delete “$Y S_2$”
Construction of generalized suffix tree \{T_1, \ldots, T_k\}

- Can use the same trick
  - construct suffix tree for \(T_1 S_1 \ldots T_k S_k\)

- Linear runtime: \(O(|T_1| + \ldots + |T_k|)\)

- 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

---

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

<table>
<thead>
<tr>
<th>mississippi</th>
<th>c</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>isissippi</td>
<td>i</td>
<td>10</td>
</tr>
<tr>
<td>isissippi</td>
<td>ippi</td>
<td>7</td>
</tr>
<tr>
<td>isissippi</td>
<td>sippi</td>
<td>5</td>
</tr>
<tr>
<td>isissippi</td>
<td>ppi</td>
<td>7</td>
</tr>
<tr>
<td>isissippi</td>
<td>ippi</td>
<td>8</td>
</tr>
<tr>
<td>isissippi</td>
<td>pl</td>
<td>9</td>
</tr>
<tr>
<td>isissippi</td>
<td>i</td>
<td>10</td>
</tr>
<tr>
<td>isissippi</td>
<td>c</td>
<td>11</td>
</tr>
</tbody>
</table>

- \(SA[i] = \text{id of the } i\text{'th smallest suffix}\)

- Number \(k\) corresponds to suffix \(T[k+1..n]\)

- Lexicographic order:
  - \(X < Xa\ldots\)
  - \(Xa\ldots < X\beta\ldots\) if \(a < \beta\)

**Pattern search from a suffix 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
- [Manber, Myers '90]: \(O(m + \log n)\) worst-case
- Store LCP table
- [Abouelhoda et al. '04]: \(O(m)\) enhanced suffix array
Enhanced suffix arrays

- [Abouelhoda,Kurtz,Ohlebuschal.'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)