Tries
Preprocessing Strings

- Preprocessing the pattern speeds up pattern matching queries
  - After preprocessing the pattern, KMP’s algorithm performs pattern matching in time proportional to the text size
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - A tries supports pattern matching queries in time proportional to the pattern size
Standard Tries

- The standard trie for a set of strings $S$ is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of $S$

Example: standard trie for the set of strings

$S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$
Analysis of Standard Tries

A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:

- $n$ total size of the strings in $S$
- $m$ size of the string parameter of the operation
- $d$ size of the alphabet
Word Matching with a Trie

- Insert the words of the text into the trie.
- Each leaf is associated with one particular word.
- Leaf stores indices where associated word begins ("see" starts at index 0 & 24, leaf for "see" stores those indices.)
Compressed Tries

- A compressed trie has internal nodes of degree at least two.
- It is obtained from a standard trie by compressing chains of "redundant" nodes.
- Ex. the "i" and "d" in "bid" are "redundant" because they signify the same word.
Compact Representation

Compact representation of a compressed trie for an array of strings:
- Stores at the nodes ranges of indices instead of substrings
- Uses $O(s)$ space, where $s$ is the number of strings in the array
- Serves as an auxiliary index structure

```
S[0] = see
S[1] = bear
S[2] = sell
S[3] = stock
S[4] = bull
S[5] = buy
S[7] = hear
S[8] = bell
S[9] = stop
```
Suffix Trie

The suffix trie of a string $X$ is the compressed trie of all the suffixes of $X$. 

```
  m i n i m i z e
  0 1 2 3 4 5 6 7
```

```
  e  i  mi
  mize nimize ze nimize ze
```
Analysis of Suffix Tries

- Compact representation of the suffix trie for a string $X$ of size $n$ from an alphabet of size $d$
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in $X$ in $O(dm)$ time, where $m$ is the size of the pattern
  - Can be constructed in $O(n)$ time

![Diagram showing a suffix trie with labels (0, 1, 2, 3, 4, 5, 6, 7) and values (7, 7, 1, 1, 0, 1, 2, 7, 6, 7, 2, 7, 6, 7).]
Encoding Trie (1)

- A code is a mapping of each character of an alphabet to a binary code-word.
- A prefix code is a binary code such that no code-word is the prefix of another code-word.
- An encoding trie represents a prefix code:
  - Each leaf stores a character.
  - The code word of a character is given by the path from the root to the leaf storing the character (0 for a left child and 1 for a right child.)

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>010</th>
<th>011</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

![Encoding Trie Diagram]

© 2004 Goodrich, Tamassia
Encoding Trie (2)

- Given a text string $X$, we want to find a prefix code for the characters of $X$ that yields a small encoding for $X$
  - Frequent characters should have short code-words
  - Rare characters should have long code-words

- Example
  - $X =$ abracadabra
  - $T_1$ encodes $X$ into 29 bits
  - $T_2$ encodes $X$ into 24 bits
Huffman’s Algorithm

Given a string $X$, Huffman’s algorithm constructs a prefix code that minimizes the size of the encoding of $X$.

It runs in time $O(n + d \log d)$, where $n$ is the size of $X$ and $d$ is the number of distinct characters of $X$.

A heap-based priority queue is used as an auxiliary structure.

**Algorithm** $HuffmanEncoding(X)$

**Input** string $X$ of size $n$

**Output** optimal encoding trie for $X$

1. $C \leftarrow distinctCharacters(X)$
2. $computeFrequencies(C, X)$
3. $Q \leftarrow$ new empty heap
4. for all $c \in C$
   - $T \leftarrow$ new single-node tree storing $c$
   - $Q.insert(getFrequency(c), T)$
5. while $Q.size() > 1$
   - $f_1 \leftarrow Q.min()$
   - $T_1 \leftarrow Q.removeMin()$
   - $f_2 \leftarrow Q.min()$
   - $T_2 \leftarrow Q.removeMin()$
   - $T \leftarrow join(T_1, T_2)$
   - $Q.insert(f_1 + f_2, T)$
6. return $Q.removeMin()$
Example

$X = \text{abracadabra}$

Frequencies

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

© 2004 Goodrich, Tamassia