Tries and suffixes trees

Alon Efrat
Computer Science Department
University of Arizona

Trie: A data-structure for a set of words

All words over the alphabet $\Sigma=\{a,b,..z\}$. In the slides, let say that the alphabet is only $\{a,b,c,d\}$.

$S$ – set of words = $\{aba, a, aca, addd\}$

Need to support the operations
• $\text{insert}(w)$ – add a new word $w$ into $S$.
• $\text{delete}(w)$ – delete the word $w$ from $S$.
• $\text{find}(w)$ is $w$ in $S$?

Future operation:
• Given text (many words) where is $w$ in the text.

• The time for each operation should be $O(k)$, where $k$ is the number of letters in $w$.

• Usually each word is associated with addition info – not discussed here.

Trie (Tree+Retrive) for $S$

A tree where each node is a struct consist
Struct node {
  char* ar;
  char flag; /* 1 if a word ends at this node. Otherwise 0 */
}

Rule:
Each node corresponds to a word $w$ ($w$ which is in $S$ if the flag is 1)

A trie - example

Note: The label of an edge is the label of the cell from which this edge exits.

Corr. to $w=d$ (not in $S$, flag=0)
Corr. to $w=dbb$ in $S$, so flag=1

Rule: Each node corresponds to a word $w$ ($w$ which is in $S$ if the flag is 1)
Finding if word $w$ is in the tree

$p$=root; $i$=0

While(1){
    • If $w[i]$ == '0' // we scanned all letters of $w$
    • then return the flag of $p$; // True/False
    • If the entry of $p$ correspond to $w[i]$ is NULL
      return false;
    • Set $p$ to be the node pointed by this entry, and set $i++$;
}

Inserting a word $w$

Recall – we need to modify the tree so find($w$) would return TRUE.

• Try to perform find($w$).
  • If runs into a NULL pointers, create new node(s) along
    the path.
  • The flag fields of all new node(s) is 0.
  • Set the flag of the last node to 1

Inserting "cbb"

$p$->ar['b'-'a']

Try to perform find("cbb").

If runs into a NULL pointers, create new node(s) along
the path.

The flag fields of all new node(s) is 0.
Set the flag of the last node to 1

Deleting a word $w$

Find the node $p$ corresponding to $w$ (using `find` operation).

• Set the flag field of $p$ to 0.

• If $p$ is dead (I.e. flag==0 and all pointers are NULL )
  then free($p$), set $p$=parent($p$) and repeat this check.
Space requirements

- Let $m$ be the sum of characters of all words in $S$
- The space required might be $\Theta(|\Sigma| m)$
- (for each letter of each words of $S$, we need an array of size $|\Sigma|$)

(Might be an issue by itself, and might slow down performances)

Heuristics for space saving

- To save some space, if $\Sigma$ is larger, there are a few heuristics we can use. Assume $\Sigma=$\{a,b..z\}.
- We use two types of nodes
  - Type “A”, which is used when the number of children of a node is more than 3
  - Type “B” is used if there are 3 or less children:
    - The “letter” of the child is also stored:

  - The rule of the flag is the same as in type “A” nodes.
  - We only store the 3 pointers, but we need to know to which letters they corresponds to

Another Heuristics – path compression

- Replace a long sequence of nodes that happens to have only a single child, with a single node (of type “pointer to string”) that keeps a point to the next node, and a point to a string.
Assume $B$ (for book) is a long text.

Want to preprocess $B$, so when a word $w$ is given, we could quickly find if it is in $B$. (incremental search)

- (as well as locations, how many etc)
- We can find it in $O(|w|)$.

Idea:
- Consider $B$ as a long string.
- Create a trie $T$ of all suffixes of $B$.
- In addition to the flag (specifying if a word ends at node), we also stored the index in $B$ where this word begins.

Example $B=\text{"aabab"}$
$S=\{\text{"aabab"}, \text{"abab"}, \text{"bab"}, \text{"ab"}, \text{"b"}\}$

To know where a word appear in $B$, we store with each node the index of the beginning of the suffix in $B$.
(we can store only the first appearance of the word in the text)

Assume $n=|B|$.

Total length of all string $\Theta(n^2)$

Size of a node is $|\Sigma|$.

So size of the tree is $\Theta(n^2 |\Sigma|)$.

Time to construct the tree $\Theta(n^2)$

Rather than a flag, we store the first index where the suffix appear

Example $B=\text{"aabab"}$
$S=\{\text{"aabab"}, \text{"abab"}, \text{"bab"}, \text{"ab"}, \text{"b"}\}$

In addition to the flag, we store the first index (in the book) where the suffix starts (in red)

Example $B=\text{"aabab"}$
$S=\{\text{"aabab"}, \text{"abab"}, \text{"bab"}, \text{"ab"}, \text{"b"}\}$
**Suffix tries on a diet**

**Def.** a shred is a path from node \( u \) to node \( v \) in the trie, consisting of nodes of outdegree 1 (except maybe the last one) and flag=0.

**Obs.** There is a contiguous part of \( B \), identical to the string the shred represents. We call this part the shred-string.

We store \( B \) itself as an array.

We use a new type of nodes, called shred-nodes, that maintain only the indexes of the first (id1) and last (id2) letters of the shred-string in \( B \).

Example for shred of “adbd”

```
B = cadbdaadbd
```

<table>
<thead>
<tr>
<th>type</th>
<th>id1</th>
<th>id2</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

**Algorithm for constructing a “thin” trie:**

Given \( B \) – create an empty trie \( T \), and insert all \( n \) suffixes of \( B \) into \( T \) — generating a trie of size \( O(n^2) \).

 Traverse the tries, and each time that a shred is seen, replace all nodes of the shred with a single shred-node.

**Observations:**

- The number of leaves of \( T \) is at most \( \leq n \) (every leaf is the end of one prefix).
- In addition there are nodes that have a single child, but their flag=1 (a suffixed have ended). We call them special nodes.
- The number of special nodes is \( \leq n \).

**Thanks for patience.**

**See you at the review**
Suffix tries on a diet - cont

Lemma: Let $T$ be a tree where each internal node has outdegree 2 or more, and $m$ leaves. Then $T$ has at most $m$ internal nodes.

Back to thin suffix tries: $T$ does not have exactly this property, but it is very close (no long shreds), so a "massaged" lemma still works, so

$\#\text{internal\_nodes} \leq \#\text{leafs\_nodes} + \#\text{special\_nodes}$

But $\#\text{leafs\_nodes} + \#\text{special\_nodes} \leq \#\text{suffixes\_of\_B} = n$

So the size of the trie is only a constant more than the size of the book.

Proof of lemma (just FYI)

Lemma: Let $T$ be a tree where each internal node has outdegree 2 or more, and $m$ leaves and $k$ internal nodes. Then $k \leq m$.

Proof: Assume true for all trees with strictly less than $m$ leaves, and assume $T$ has $m$ leaves.

Find a leaf $u$ whose distance from root is maximum. Assume it has exactly one sibling $v$. Note that $v$ is a leaf (why?). Let $w$ be their common parent.

Remove both $u$ and $v$ from $T$. Let $T'$ be the resulting tree.

Let $k', m'$ denote the number of internal nodes and leaves in $T'$. Now in $T'$

- $w$ is a leaf.
- $m' = m - 1$.
- $k' = k - 1$.
- The outdegree of every internal node $\geq 2$

From induction, $k' \leq m'$. Hence $k \leq m$. 

So the size of the trie is only a constant more than the size of the book.