Tries and suffix trees

Alon Efrat
Computer Science Department
University of Arizona

Trie: A data-structure for a set of words

All words over the alphabet \( \Sigma = \{a, b, \ldots, z\} \).
In the slides, the alphabet is only \( \{a, b, c, d\} \).
\( S \) – set of words = \{a, aba, a, aca, addd\}.
Need to support the operations
• insert(\( w \)) – add a new word \( w \) into \( S \).
• delete(\( w \)) – delete the word \( w \) from \( S \).
• 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[4] *ar;
    • char flag; /* 1 if a word ends at this node. Otherwise 0 */
  }

Rule:
Each node corresponds to a word \( w \).
\( w \subseteq S \) iff flag=1

A trie - example

The dictionary contains \( S = \{a, b, dbb\} \)
Corr. To \( w = "d" \)
Corr. To \( w = "dbb" \)
The label of an edge is the label of the cell from which this edge exits
Corr. To \( w = "db" \) (not in \( S \), flag=0)
A quick reminder from Java/C

the when we write 'a', it means “the ascii value of ‘a’.

For example, 'A'=65, 'B'=66,.. 'Z'=90, 'a'=97 etc

This means ‘d’-'a'=d,

---

Finding if word \( w \) is in the tree

\( p = \text{root}; \ i = 0 \ // \text{remember - each string ends with } \backslash 0 \)

While(1){

  • If \( w[i] == \backslash 0 \) //we have scanned all letters of \( w \)
    then return the flag of \( p \); \textbf{else}
  • If \( (p.a[w[i] - 'a']) == \text{NULL} \ //\text{the entry of } p \text{ correspond to } w[i] \text{ is NULL} \)
    return \textbf{false};
  • \( p = (p.a[w[i] - 'a']) \) //Set \( p \) to be the node pointed by this entry
  • \( i++; \)
}

---

Inserting a word \( w \)

• Try to perform find(\( w \)).
  • If runs into a NULL pointers, create new nodes along the path.
  • The flag fields of all new nodes 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 \( p=\text{parent}(p) \) and repeat this check.
Heuristics for saving space

- The space required is $\Theta(|\Sigma| |S|)$.
- 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:

Note – the letters are not stores explicitly

Another Heuristics – path compression

- Replace a long sequence of nodes, all having only one a single child, with a single node (of type “pointer to string”) that maintains
  - a point to the next node,
  - a point to the string.

Heuristics for space saving

- 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.

Suffix tree.

- Assume $B$ (for book) is a very long text.
- Want to preprocess $B$, so when a word $w$ is given, we can quickly find if it is in $B$.
- 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 $B=“aabab”$
  - $S=\{“aabab”, “abab”, “bab”, “ab”, “b”\}$
Suffix tree.

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

![Diagram of suffix tree]

To know where a word $w \in S$ appears in $B$, we store with the node of the starting_index of $w$ in $B$. We store only the first appearance of the word in the text (shown in brown).

Size of suffix tree

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

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 $\Theta(n^2)$

We can save some space.

Suffix tries on a diet - cont

**Def:** a thread 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 contagious part of $B$, identical to the string the shred represents. We call this part the shred-string.

We stores the book $B$ itself as an array.

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

<table>
<thead>
<tr>
<th>type</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>id1</th>
<th>id2</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>B=\text{&quot;cadbdaadbd&quot;}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 $\Theta(n^2)$.

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

• Clearly the use of thread-nodes saves some—but can we prove something?

Observations: Every leaf of $T$ must be the end of some prefix of $B$. So the number of number of leaves of $T$ is $\leq n$. ($n$ denotes the book size)

• To bound the size of $T$, we will need to bound the number of internal nodes.

Observations:

○ $T$ might contain special nodes whose $\text{flag}=1$ (a suffix terminates at these nodes).

○ The number of special nodes is also $\leq n$ (since this is the number of suffixes).

• What about other internal nodes of $T$?

Back to compressed suffix trees

Back to thin suffix tries $T$ created for a book $B$ with $n$ letters.

- $T$ has $\leq n$ special nodes (with $\text{flag}=1$) and
- $T$ has $\leq n$ leaves (every leaf is the end of a suffix of $B$)
- Every other nodes has $\geq 2$ children. (with $\text{flag}=1$). Applying the children-blessed Lemma in this case, implies that the total number of internal nodes $\leq 2n$.

• Conclusion: The number of nodes in $T$ is $\leq 3n$ (much better than the uncompressed version that could have $\Theta(n^2)$ nodes.

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

The “children-blessed Lemma”

We say that a tree $T$ is children-blessed if every node is either a leaf or has $\geq 2$ children.

Let $T$ be a tree with $m$ leaves. We use the following notation:

- $\text{leaves}(T)$ denote the number of leaves in $T$.
- $\text{nodes}(T)$ denote the # of nodes in $T$.
- $\text{internal}(T)$ denote the # of internal in $T$.

Children-blessed Lemma: If $T$ is a children-blessed tree, then $\text{internal}(T) \leq \text{leaves}(T)$. That is, $T$ has more leaves than internal nodes.

Proof by induction on $m$ (the number of leaves in $T$)

Base case: $m=1$. A children-blessed tree $T$ that has only one leaf $u$ must have zero internal nodes. If $u$ has a parent, then this parent is internal but $u$ is the only child. So the base case is proven the induction base case.

Induction step. Pick some integer $m \geq 2$. Assume that we have proven the lemma for every c.b. tree that has $\leq m$ leaves. and let $T$ be a children-blessed tree that has $m+1$ leaves. Need to show $\text{internal}(T) \leq m+1$. Pick an arbitrary leaf $u$ of $T$, and let $v = \text{parent}(u)$. Now we have two cases, depending on the number of siblings of $u$:

1. Case 1: $u$ has at least 2 siblings. Create a tree $T'$ by deleting $u$ from $T$.

- $T'$ is still children-blessed.
- $\text{internal}(T') = \text{internal}(T)$ but $\text{leaves}(T') = \text{leaves}(T) + 1$.

Since $m = \text{nodes}(T') - 1$ and our assumption is that the lemma has been proven for all trees with $\leq m$ leaves, we know that $\text{internal}(T') \leq \text{leaves}(T')$, implying that $\text{internal}(T) \leq \text{leaves}(T)$.

2. Case 2: $u$ has only one sibling $v$. Let $p = \text{parent}(u)$. Create a tree $T'$ by deleting both $u$, and $v$ from $T$.

- In $T'$, stopped being an internal node, and is now a leaf. $T'$ is still children-blessed.
- $\text{internal}(T') = \text{internal}(T) - 1$
- $\text{leaves}(T') = \text{leaves}(T)$, so we could use the induction hypothesis that $\text{internal}(T') \leq \text{leaves}(T')$, therefore $\text{internal}(T) \leq \text{leaves}(T)$. This ends the proof.

Summary, and potential points of confusions

1. A trie stores a set of strings $(x_1, x_2, ..., x_n)$. The memory need is approximately $|x_1| + |x_2| + |x_3| + ... + |x_n|$ in the worst case. Here $|x_i|$ is the number of character in $x_i$.

2. An uncompressed suffix tree is a trie, but the input dictionary consists of all suffixes of a book $B$, and each node also stores where the corresponding suffix appears in $B$. The memory needed for an uncompressed suffix tree is $\Theta(n^2)$. (so as bad as $n^2$)

3. Path compression identifies in the trie long threads of nodes, each point to the next, and each has only one child. Such a thread, containing say $k$ nodes, could be replaced by a single “fancy” node. However, 3.1. In a regular trie, this node must still store $k$ character, so its size could be very large

3.2. In a suffix tree, this node only need to stores a pointer to the book, and the length of this thread. So only $O(1)$ memory

4. Path compression shrinks the size of the uncompressed suffix tree from $\Theta(n^2)$ to $\Theta(n)$. This is easily the difference between being practical to useless. We used the children-blessed lemma to show the size of the compressed suffix tree.