Tries & Suffix Tries
Ben Langmead

Please sign guestbook (www.langmead-lab.org/teaching-materials) to tell me briefly how you are using the slides. For original Keynote files, email me (ben.langmead@gmail.com).
A trie ("try") is a tree representing a collection of strings (keys): the smallest tree such that

Each edge is labeled with a character $c \in \Sigma$

For given node, at most one child edge has label $c$, for any $c \in \Sigma$

Each key is "spelled out" along some path starting at root

Helpful for implementing a set or map when the keys are strings
Tries

Keys: instant, internal, internet

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>1</td>
</tr>
<tr>
<td>internal</td>
<td>2</td>
</tr>
<tr>
<td>internet</td>
<td>3</td>
</tr>
</tbody>
</table>

Smallest tree such that:

Each edge is labeled with a character $c \in \Sigma$

For given node, at most one child edge has label $c$, for any $c \in \Sigma$

Each key is “spelled out” along some path starting at root
How do we check whether “infer” is in the trie?

Start at root and try to match successive characters of “infer” to edges in trie

**Diagram:**
- Start at the root node.
- Move down the trie by following the edges corresponding to each character of “infer”.
- The path from the root to the nodes labeled “i”, “n”, “s”, “t” is successful up to this point.
- At the “t” node, there is no edge for “e”, indicating that “f” is not in the trie.

**Result:** oops, no “f”
Tries

Matching “interesting”
Matching “insta”

No value associated with node, so “insta” wasn’t a key
Tries

Matching “instant”

It’s a key!
Tries

Checking for presence of key $P$, where $|P| = n$ traverses $\leq n$ edges

If total length of all keys is $N$, trie has $\leq N$ edges
How to represent edges between a node and its children?

*Map* (from characters to child nodes)

Idea 1: Hash table
Idea 2: Sorted lists

Assuming hash table, it’s reasonable to say querying with $P, |P| = n$, is $O(n)$ time
Tries

Could use trie to represent $k$-mer index. Map $k$-mers to offsets where they occur

<table>
<thead>
<tr>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>4</td>
</tr>
<tr>
<td>ag</td>
<td>8</td>
</tr>
<tr>
<td>at</td>
<td>14</td>
</tr>
<tr>
<td>cc</td>
<td>12</td>
</tr>
<tr>
<td>cc</td>
<td>2</td>
</tr>
<tr>
<td>ct</td>
<td>6</td>
</tr>
<tr>
<td>gt</td>
<td>18</td>
</tr>
<tr>
<td>gt</td>
<td>0</td>
</tr>
<tr>
<td>ta</td>
<td>10</td>
</tr>
<tr>
<td>tt</td>
<td>16</td>
</tr>
</tbody>
</table>
Tries: implementation

Tries: alternatives

Tries aren’t the only way to encode sets or maps over strings using a tree.

E.g. ternary search tree:

![Ternary Search Tree Diagram]

*Figure 2. A ternary search tree for 12 two-letter words*

Indexing with suffixes

We studied indexes built over substrings of $T$

Different approach is to index *suffixes* of $T$. This yields surprisingly economical & practical data structures:

- **Suffix Tree**
- **Suffix Array**
- **FM Index**
Suffix trie

Build a trie containing all suffixes of a text $T$

$$T: \text{GTTATAGCTGATCGCGGCGTAGCGG}\$$

$$m(m+1)/2\text{ chars}$$
Suffix trie

First add special terminal character $ to the end of $T$

$ is a character that does not appear elsewhere in $T$, and we define it to be less than other characters ($\$ < A < C < G < T$)

$ ensures a familiar rule: e.g. “as” comes before “ash” in the dictionary.
$ also guarantees no suffix is a prefix of any other suffix.

\[
T: \text{GT'TATAGCTGATCGCGGCGTAGCGG}\$ \\
GTTATAGCTGATCGCGGCGTAGCGG$
\text{GT'TATAGCTGATCGCGGCGTAGCGG}$
\text{TTATAGCTGATCGCGGCGTAGCGG}$
\text{TATAGCTGATCGCGGCGTAGCGG}$
\text{ATAGCTGATCGCGGCGTAGCGG}$
\text{TAGCTGATCGCGGCGTAGCGG}$
\text{AGCTGATCGCGGCGTAGCGG}$
\text{GCTGATCGCGGCGTAGCGG}$
\text{CTGATCGCGGCGTAGCGG}$
\text{TGATCGCGGCGTAGCGG}$
\text{GATCGCGGCGTAGCGG}$
\text{ATCGCGGCGTAGCGG}$
\text{TCGCGGCGTAGCGG}$
\text{CGCGGCGTAGCGG}$
\text{GC GGCGTAGCGG}$
\text{CGGGCGTAGCGG}$
\text{CGGGCGTAGCGG}$
\text{CGGGCGTAGCGG}$
\text{CGGGCGTAGCGG}$
\text{CGGGCGTAGCGG}$
\text{CGGGCGTAGCGG}$
\text{CGGGCGTAGCGG}$
\text{CGGGCGTAGCGG}
**Suffix trie**

*T: aba$

What’s the suffix trie?
def __init__(self, t):
    """ Make suffix trie from t """
    t += '$'                # add terminator
    self.root = {}
    for i in range(len(t)):  # for each suffix
        cur = self.root
        for c in t[i:]:        # for each character in i'th suffix
            if c not in cur:
                cur[c] = {}       # add outgoing edge if necessary
            cur = cur[c]         # follow the edge and continue

**Suffix trie**

*T: abaaba$*

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf.

Would this still be the case if we hadn’t added $?$
Suffix trie

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf.

Would this still be the case if we hadn’t added $\$ $?  **No**

$T$: abaaba$
Suffix trie

Think of each node as having a **label**, spelling out characters on path from root to node.
**Suffix trie**

How do we check whether a string $S$ is a substring of $T$?

Note: Each of $T$'s substrings is spelled out along a path from the root.

*Every substring is a prefix of some suffix* of $T$.

Start at the root and follow the edges labeled with the characters of $S$

If we “fall off” the trie -- i.e. there is no outgoing edge for next character of $S$, then $S$ is not a substring of $T$

If we exhaust $S$ without falling off, $S$ is a substring of $T$
**Suffix trie**

How do we check whether a string $S$ is a substring of $T$?

Note: Each of $T$'s substrings is spelled out along a path from the root.

*Every substring is a prefix of some suffix* of $T$.

Start at the root and follow the edges labeled with the characters of $S$

If we “fall off” the trie -- i.e. there is no outgoing edge for next character of $S$, then $S$ is not a substring of $T$

If we exhaust $S$ without falling off, $S$ is a substring of $T$

$S = abaaba$

Yes, it’s a substring
**Suffix trie**

How do we check whether a string $S$ is a substring of $T$?

Note: Each of $T$'s substrings is spelled out along a path from the root.

*Every substring is a prefix of some suffix* of $T$.

Start at the root and follow the edges labeled with the characters of $S$

- If we “fall off” the trie -- i.e. there is no outgoing edge for next character of $S$, then $S$ is not a substring of $T$
- If we exhaust $S$ without falling off, $S$ is a substring of $T$

$T = \text{abaaba}$

$S = \text{baabb}$

No, not a substring
def follow_path(self, s):
    """ Follow path given by characters of s. Return node at end of path, or None if we fall off. """
    cur = self.root
    for c in s:
        if c not in cur:
            return None  # no outgoing edge on next character
        cur = cur[c]  # descend one level
    return cur

def has_substring(self, s):
    """ Return true if s appears as a substring of t """
    return self.follow_path(s) is not None
**Suffix trie**

How do we check whether a string \( S \) is a **suffix** of \( T \)?

Same procedure as for substring, but additionally check terminal node for $ child

\[ S = baa \]

**Not** a suffix
Suffix trie

How do we check whether a string $S$ is a suffix of $T$?

Same procedure as for substring, but additionally check terminal node for $\$ \text{ child}$

$S = \text{aba}$ is a suffix
def has_suffix(self, s):
    """Return true if s is a suffix of t """
    node = self.folow_path(s)
    return node is not None and '$' in node
Su‌ffix trie

How do we count the **number of times** a string $S$ occurs as a substring of $T$?

Follow path labeled with $S$. If we fall off, answer is 0. If we end up at node $n$, answer equals # of leaves in subtree rooted at $n$.

Leaves can be counted with depth-first traversal.
Suffix trie

How do we find the **longest repeated substring** of $T$?

Find the deepest node with more than one child
Suffix trie

How does the suffix trie grow with $|T| = m$?
Suffix trie

How does the suffix trie grow with $|T| = m$?

Is there a class of string where the number of suffix trie nodes grows linearly with $m$?

Yes: a string of $m$ a’s in a row ($a^m$)

- 1 Root
- $m$ nodes with "a" edge to parent
- $m + 1$ nodes with incoming $\$ edge
- $2m + 2$ nodes
Suffix trie

How does the suffix trie grow with $|T| = m$?

Is there a class of string where the number of suffix trie nodes grows with $m^2$?

Yes: $a^n b^n$ where $2n = m$

- 1 root
- $n$ nodes along “b chain,” right
- $n$ nodes along “a chain,” middle
- $n$ chains of $n$ “b” nodes hanging off “a chain” ($n^2$ total)
- $2n + 1$ leaves (not shown)

$n^2 + 4n + 2$ nodes, where $m = 2n$
Suffix trie: upper bound on size: idea 1

Recall our function for build building a suffix trie

```python
def __init__(self, t):
    """ Make suffix trie from t """
    t += '$'
    self.root = {}
    for i in range(len(t)):
        cur = self.root
        for c in t[i:]:
            if c not in cur:
                cur[c] = {}
            cur = cur[c]
```

Adds 1 node, runs at most $m(m+1)/2$ times
Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length
Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length

Actual growth *much* closer to worst case than to best!