String Algorithms and Data Structures

Tries

CS 199-225
Brad Solomon

February 27, 2023
Exact pattern matching w/ Boyer-Moore

As seen in HW: sub-linear time in practice

Make lookup tables

Boyer-Moore $\approx O(|P| + |T|)$

Find instances of $P$ in $T$
Preprocessing: Live chat streams

**Patterns:** banned phrases

**Text:** Chat messages
Preprocessing: Live chat streams

Preprocess

Find instances of $P$ in $T$

Amortize cost of preprocessing $P$ over many $T$

Boyer-Moore

Thanks for the help!

I don’t understand that…

You are a !@#$!% teacher
Exact pattern matching \textit{w/ indexing}

Conventionally $T \gg P$:

\begin{itemize}
  \item $P_1$, $P_2$
  \item $P_3$, $P_4$
\end{itemize}

Preprocess (index)

Search Index

Find instances of $P$ in $T$

\textit{Amortize} cost of preprocessing $T$ over many $P$
Preprocessing: Libraries

**Patterns:** Book of interest  
**Text:** All books in library
Preprocessing: Libraries

Preprocess the library by *indexing* all the books
Preprocessing: Libraries

Given full library, built an index once* that is re-used

List of all library books

Preprocess (index)

Search Index

Find instances of $P$ in $T$
# Preprocessing: Glossaries

What method of preprocessing is this?

<table>
<thead>
<tr>
<th>Text: All text in the book</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Patterns:</strong> Key terms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Glossary</th>
<th>Page References</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Glossary</strong>: Nest site hunting</td>
<td>482–87</td>
</tr>
<tr>
<td>honeypot ants, see Myrmecocystus</td>
<td></td>
</tr>
<tr>
<td>hormones, 106–9</td>
<td></td>
</tr>
<tr>
<td>see also exocrine glands</td>
<td></td>
</tr>
<tr>
<td>house (nest site) hunting, 482–92</td>
<td></td>
</tr>
<tr>
<td>Hymenopera (general), xvi</td>
<td></td>
</tr>
<tr>
<td>haplodiploid sex determination, 20–22</td>
<td></td>
</tr>
<tr>
<td><em>Hypoponera</em> (ants), 194, 262, 324, 388</td>
<td></td>
</tr>
<tr>
<td>inclusive fitness, 20–23, 29–42</td>
<td></td>
</tr>
<tr>
<td>information measurement, 251–52</td>
<td></td>
</tr>
<tr>
<td>intercastes, 388–89</td>
<td></td>
</tr>
<tr>
<td>see also ergatogynes; ergatoid queens; gamergates</td>
<td></td>
</tr>
<tr>
<td><em>Iridomyrmex</em> (ants), 266, 280, 288, 321</td>
<td></td>
</tr>
<tr>
<td>Isoptera, see termites</td>
<td></td>
</tr>
<tr>
<td>juvenile hormone, caste, 106–9, 372</td>
<td></td>
</tr>
<tr>
<td>kin recognition, 293–98</td>
<td></td>
</tr>
<tr>
<td>kin selection, 18–19, 23–24, 28–42, 299, 386</td>
<td></td>
</tr>
<tr>
<td><em>Macrotermes</em> (termites), 59–60</td>
<td></td>
</tr>
<tr>
<td>male recognition, 298</td>
<td></td>
</tr>
<tr>
<td>mass communication, 62–63, 214–18</td>
<td></td>
</tr>
<tr>
<td>mating, multiple, 155</td>
<td></td>
</tr>
<tr>
<td>maze following, 119</td>
<td></td>
</tr>
<tr>
<td><em>Megalomyrmex</em> (ants), 457</td>
<td></td>
</tr>
<tr>
<td><em>Mega ponerinae</em> (ants), see Pachycondyla</td>
<td></td>
</tr>
<tr>
<td>Melipona (stingless bees), 129</td>
<td></td>
</tr>
<tr>
<td><em>Melophorus</em> (ants), repletes, 257</td>
<td></td>
</tr>
<tr>
<td>memory, 117–19, 213</td>
<td></td>
</tr>
<tr>
<td><em>Messor</em> (harvester ants), 212, 232</td>
<td></td>
</tr>
<tr>
<td>mind, 117–19</td>
<td></td>
</tr>
<tr>
<td>Monomorium, 127, 212, 214, 216–17, 292</td>
<td></td>
</tr>
<tr>
<td>motor displays, 235–47</td>
<td></td>
</tr>
<tr>
<td>mound-building ants, 2</td>
<td></td>
</tr>
<tr>
<td>multilevel selection, 7, 7–13, 24–29</td>
<td></td>
</tr>
<tr>
<td>mutilation, ritual, 366–73</td>
<td></td>
</tr>
<tr>
<td>mutualism, see symbioses, ants</td>
<td></td>
</tr>
<tr>
<td><em>Mymaromyra</em> (fossil ants), 318</td>
<td></td>
</tr>
<tr>
<td>Myopias (ants), 326</td>
<td></td>
</tr>
</tbody>
</table>
Preprocessing: Glossaries

Memory

*kin recognition*

mind

Biology Textbook

Preprocess (index)

Search Index

Find instances of $P$ in $T$

Glossary built on total contents $T$, useful for multiple $P$
Exact pattern matching \textit{w/indexing}

The human genome (~3 billion)

1 Billion reads (~100)

Preprocess (index) \approx O(|T|)

Search Index \approx O(|P|)

Find instances of $P$ in $T$

Amortize cost of preprocessing $T$ over many $P$
Exact pattern matching with indexing

What information from $T$ do we need to search for $P$?
Preprocessing for exact pattern matching

\[ T: \text{CGTGC} \]

\[ \text{Search}(P, T): \]

\[ P: \]

\[ \text{Search}(P, T): \]

\[ P: \]

\[ \text{Search}(P, T): \]
Preprocessing for exact pattern matching

\[ T: \text{CGTGC} \]

A substring \( S \)  

The position of \( S \) in \( T \)
Preprocessing for exact pattern matching

\( T : \) C G T G C

\[
\begin{array}{c}
|T| \\
C \\
G \\
T \\
G \\
C \\
|T-1| \\
CG \\
GT \\
TG \\
GC \\
|T-2| \\
C G T \\
G T G \\
T G C \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Preprocessing for exact pattern matching

\( T: \) C G T G C

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \frac{|T|(|T| + 1)}{2} \]
Preprocessing for exact pattern matching

Because our keys are strings, this is sometimes possible!

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

We want to search in $O(|P|)$ without $O(|T|^2)$ space!
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

\( T: C G T G C \)
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T: C\ G\ T\ G\ C$

$G$
$G\ T$
$G\ T\ G$
$G\ T\ G\ C$

root:
Preprocessing for exact pattern matching

Strings consist of individual characters!

… and these characters can overlap:
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

\[ T: C G T G C \]

\[ G \]

\[ G C \]
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T: C G T G C$

```
root: G -> T -> G -> C
   |     |      |
  C --> G --> T -- C
```

```
C
C

C
T
T
G
G
C

C
T
G
C

G
C
G
C
```
String indexing with Tries

**Trie:** A rooted tree storing a collection of (key, value) pairs

Keys:  

- instant 1  
- internal 2  
- internet 3  

Values: 

- 1  
- 2  
- 3  

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
String indexing with Tries

**Trie:** A rooted tree storing a collection of (key, value) pairs

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</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>

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
String indexing with Tries

**Trie:** A rooted tree storing a collection of (key, value) pairs

Keys: \( \text{instant} \quad 1 \)

Values:

\( \text{internal} \quad 2 \)

\( \text{internet} \quad 3 \)

Where should I store the value 1?
String indexing with Tries

**Trie:** A rooted tree storing a collection of (key, value) pairs

Keys: 

- instant 1
- internal 2
- internet 3
- internets 4

Values: 

1 2 3 4

Each key’s value is stored at the last node in the path
The Node Implementation

Each node in my trie has $\leq |\Sigma|$ edges!

Each edge is a (potentially NULL) pointer.

How can we encode this?
The Node Implementation

Each node in my trie has $\leq |\Sigma|$ edges!

Each edge is a (potentially NULL) pointer.

1) Static Array

2) Dynamically-sized Dictionary (std::map)
class NaryTree
{
    public:
        struct Node {
            std::vector<int> index;
            std::map<char, Node*> children;

            Node(std::string s, int i)
            {
                if(s.length() > 0 ){
                    children[s[0]] = new Node(s.substr(1), i);
                } else {
                    index.push_back(i);
                }
            }

        };

    protected:
        Node* root;
};
**Trie Node Implementation**

```cpp
class NaryTree
{
  public:
    struct Node {
      std::vector<int> index;
      std::map<char, Node*> children;

      Node(const std::string& s, int i)
      {
        if(s.length() > 0)
        {
          children[s[0]] = new Node(s.substr(1), i);
        } else {
          index.push_back(i);
        }
      }
    }

  protected:
    Node* root;
};
```

What if we have more than one string?
Trie Node Implementation

```cpp
main.cpp

NaryTree myT;
myTree.print();
myTree.insert("AB", 0);
myTree.print();
myTree.insert("ABA", 1);
myTree.print();
myTree.insert("ABB", 2);
myTree.print();
myTree.insert("BAB", 3);
myTree.print();
myTree.insert("BBB", 4);
myTree.print();
```
Trie Node Implementation

```cpp
main.cpp

NaryTree myT;
myTree.print();
myTree.insert("AB",0);
myTree.print();
myTree.insert("ABA",1);
myTree.print();
myTree.insert("ABB",2);
myTree.print();
myTree.insert("BAB",3);
myTree.print();
myTree.insert("BBB",4);
myTree.print();
```

```plaintext
+--A--o
  +--B--o
  ```
Trie Node Implementation

```cpp
main.cpp

```

NaryTree myT;
myTree.print();
myTree.insert("AB", 0);
myTree.print();
myTree.insert("ABA", 1);
myTree.print();
myTree.insert("ABB", 2);
myTree.print();
myTree.insert("BAB", 3);
myTree.print();
myTree.insert("BBB", 4);
myTree.print();
```

Former leaf node, still holds value

```
struct Node {
    std::vector<int> index;
    std::map<char, Node*> children;
}
```
NaryTree myT;
myTree.print();
myTree.insert("AB", 0);
myTree.print();
myTree.insert("ABA", 1);
myTree.print();
myTree.insert("ABB", 2);
myTree.print();
myTree.insert("BAB", 3);
myTree.print();
myTree.insert("BBB", 4);
myTree.print();
Trie Node Implementation

main.cpp

```cpp
NaryTree myT;
myTree.print();
myTree.insert("AB",0);
myTree.print();
myTree.insert("ABA",1);
myTree.print();
myTree.insert("ABB",2);
myTree.print();
myTree.insert("BAB",3);
myTree.print();
myTree.insert("BBB",4);
myTree.print();
```
NaryTree myT;
myTree.print();
myTree.insert("AB",0);
myTree.print();
myTree.insert("ABA",1);
myTree.print();
myTree.insert("ABB",2);
myTree.print();
myTree.insert("BAB",3);
myTree.print();
myTree.insert("BBB",4);
myTree.print();
void NaryTree::insert(const std::string& s, int i) {
    insert(root, s, int i);
}

void NaryTree::insert(Node*& node, const std::string & s, int i) {
    // If we're at a NULL pointer, we make a new Node
    if (node == NULL) {
        node = new Node(s, i);
    } else {
        if(s.length() > 0 ){
            if(node->children.count(s[0]) > 0){
                insert(node->children[s[0]],s.substr(1), i);
            }else{
                node->children[s[0]] = new Node(s.substr(1), i);
            }
        } else{
            node->index.push_back(i);
        }
    }
}
Assignment 5: a_narytree

Learning Objective:

**Store all substrings in a trie using NaryTree implementation**

Implement exact pattern matching using this trie

Consider: How many insertions are we doing for each string? Is there a better or faster way to do this?
NaryTree myT;
myTree.insert("AB",0);
myTree.insert("AB",2);
myTree.print();
Searching a Trie

Given \( P \), search the trie for keys and return values

Pattern: \( Infer \)

\( Infer \)

\( Infer \)

\( Infer \)

Lets break that down using *recursion*:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to (1)
3. If mismatch, \( P \) is not a key!
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: interesting
  interesting
  interesting

Let's break that down using recursion:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to (1)
3. If mismatch, $P$ is not a key!
Searching a Trie

Given \( P \), search the trie for keys and return values

Pattern: \( \text{ina} \) \( \text{insta} \)

Let's break that down using *recursion*:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to (1)
3. If mismatch, \( P \) is not a key!
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: $\text{insta}$

Lets break that down using recursion:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5. Set pattern equal to remainder
   2.5. Go back to (1)
3. If mismatch, $P$ is not a key!

“Insta” is NOT a key!
There’s no value here!
String indexing with Tries

A rooted tree storing a collection of (key, value) pairs

Keys: \hspace{1cm} Values:

\begin{itemize}
  \item \texttt{instant} \hspace{1cm} 1
  \item \texttt{internal} \hspace{1cm} 2
  \item \texttt{internet} \hspace{1cm} 3
\end{itemize}

The trie is structured 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.

Each key’s value is stored at the last node in the path.
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: `insta`

```
  insta
```

Lets break that down using recursion:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to (1)
3. If mismatch, $P$ is not a key!

```
Pattern: insta
```

```
  insta
```

“Insta” is NOT a key!
There’s no value here!

Given $P$, search the trie for keys and return values.
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: `instant`

Let's break that down using recursion:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to (1)
3. If mismatch, $P$ is not a key!

Given $P$, search the trie for keys and return values
Assignment 5: a_narytree

Learning Objective:

Store all substrings in a trie using NaryTree implementation

Implement exact pattern matching using this trie

Consider: How could we search the trie if we are only allowed to store one value in each node [instead of a vector of them]?
**Preprocessing for exact pattern matching**

**$T$: C G T G C**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Preprocessing for exact pattern matching

\[ T: C \ G \ T \ G \ C \]

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

We can do exact pattern matching in \( O(P) \) time!
Preprocessing for exact pattern matching

Let us consider the pattern $T: \text{CGTGC}$.

We are storing $\frac{|T|(|T|+1)}{2}$ values.

We had to do $\frac{|T|(|T|+1)}{2}$ insertions.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Preprocessing for exact pattern matching

If only there was a way…

to insert fewer strings
to store fewer values
If only there was a way…

to insert fewer strings

to store fewer values

to be even more efficient!